

GroupWise[®] developer's guide

Novell Research

www.novell.com

SECTION 1 & 3



Michael Risch,
Sean Kirkby,
Bob Good,
Steve Hughes

CONTENTS

Section 1: GroupWise Object API

4 Chapter 1-9

Section 2: GroupWise Tokens

(not available yet)

Section 3: GroupWise C3POs

169 Chapter 16-19

**Section 4: Administrative Object
API**

(not yet available)

Novell.

Novell Research

Publishers of AppNotes®
Novell's Technical Journal for
Implementing, Managing,
and Programming to one Net

April 15, 2002

464-000063-013

Editor-in-Chief Gamal B. Herbon

Managing Editor Ken Neff

Senior Editor Edward Liebing

Senior Research Engineer Kevin Burnett

Editor Rebecca Rousseau

Graphics Wes Heaps

Online Production Robert Rodriguez

This *eBook* was produced using Adobe
FrameMaker 6, Illustrator, Photoshop, and
Streamline, QuarkXPress, WebWorks
Publisher Professional, Adobe Acrobat 5, and
HP LaserJet printers.

Editorial Comments

Direct any comments or suggestions about the
content of this eBook to:

Gamal B. Herbon
Novell Research
Novell, Inc. MS PRV A-231
1800 S. Novell Place
Provo, Utah 84606 USA
Voice (801) 861-6504
Fax (801) 861-4123
E-mail gherbon@novell.com

On the Web

The *GroupWise Object API Developer's Guide* eBook is
accessible on the World Wide Web at [http://www.novell.com/
research](http://www.novell.com/research).

Permissions

You must obtain permission before reproducing any material
from the *GroupWise Developer's Guide* in any form. To quote
from or reprint any portion, send a written request to Novell
Research Editor-in-Chief, 1800 S. Novell Place, Provo, UT,
84606, or fax to (801) 861-4123.

Copyright © 2002 by Novell, Inc., Provo, Utah. All rights
reserved.

Novell, Inc. makes no representations or warranties with respect
to the contents or use of these eBooks or of any of the third-
party products discussed in the eBook. Novell reserves the right
to revise these eBooks and to make changes in their content at
any time, without obligation to notify any person or entity of
such revisions or changes. These eBooks do not constitute an
endorsement of the third-party product or products that were
tested. Configuration(s) tested or described may or may not be
the only available solution. Any test is not a determination of
product quality or correctness, nor does it ensure compliance
with any federal, state or local requirements. Novell does not
warranty products except as stated in applicable Novell product
warranties or license agreements.

Introduction

July 2002

Dear Readers:

Welcome to this second edition of the AppNotes/DeveloperNet University GroupWise Developer's Guide. Many of you have liked what we have provided so far, and we are continuing to bring you the material you need. As we continue to release new sections as part of this eBook, please be patient and provide us with feedback on the sections as they come out, and I think the end product will be one that will be quite useful in your development efforts regarding GroupWise. Now, a little about the book.

The GroupWise APIs provide a rich set of application tools that can help developers access the power of GroupWise. This book was written to help application developers harness this power. Through a combination of numerous examples and detailed discussions not available elsewhere, its aim is to simplify and take the mystery out of GroupWise development.

The book is divided into four main sections. Chapters 1-9 discuss the GroupWise Object API, which gives developers access to the GroupWise data store. Chapters 10-15 present a discussion of GroupWise tokens. These tokens allow developers to “throw” and “catch” many of the same actions that take place within the GroupWise client itself. Chapters 16-19 discuss GroupWise C3POs, which give developers the ability to modify the GroupWise client. C3POs also provide the capability to catch and handle GroupWise events and commands. Finally, Chapters 20 and beyond present a quick survey of the Administrative Object API. This API gives someone with administrative privileges the ability to make GroupWise system level changes, such as adding and deleting GroupWise users. This book should be helpful to all GroupWise developers who use any of these API sets.

Because Visual Basic, Delphi, and C++ are all languages used by GroupWise developers, this book will include examples in all three languages. An emphasis is given to Visual Basic and Delphi, since these are the language most often used by GroupWise developers.

This release presents the first and third sections, (consisting of Chapters 1-9 and 16-19). Other chapters will be grouped together logically and will be released as we finish them. Check our web site for updated versions.

I acknowledge the contributions of the authors, Michael Risch, Sean Kirkby, Bob Good, and Steve Hughes. I also want to acknowledge and thank John Cox and Glade Monson of Novell's Worldwide Developer Support organization for their efforts.

Of course, I acknowledge the hard work of the AppNotes and DeveloperNet University team in editing, and producing this book.

I look forward to your feedback on this book and the format. Let me know how you like and what we can do better.

Gamal B. Herbon
Editor-in-Chief

GroupWise Developer's Guide

by Michael Risch, Sean Kirkby, Bob Good, and Steve Hughes

Section 1: GroupWise Object API

Gives developers access to the GroupWise data store.

- 4 *Chapter 1: Introducing the Object API*
- 26 *Chapter 2: Using the Account Object*
- 36 *Chapter 3: Understanding Folder and Trash Related Objects*
- 60 *Chapter 4: Message Collections*
- 75 *Chapter 5: Understanding Message and Message Related Objects*
- 97 *Chapter 6: Understanding Document and Document Related Objects*
- 118 *Chapter 7: Understanding Address and AddressBook Objects*
- 127 *Chapter 8: Understanding Field and Field Related Objects*
- 157 *Chapter 9: Understanding Filter and Query Related Objects*

Section 2: GroupWise Tokens *(not yet available)*

Tokens allow developers to “throw” and “catch” many of the same actions that take place within the GroupWise client itself.

Chapter 10: (not yet available)

Chapter 11: (not yet available)

Chapter 12: (not yet available)

Chapter 13: (not yet available)

Chapter 14: (not yet available)

Chapter 15: (not yet available)

Section 3: GroupWise C3POs *(not yet available)*

C3POs give developers the ability to modify the GroupWise client and also provide the capability to catch and handle GroupWise events and commands.

169 *Chapter 16: Overview*

180 *Chapter 17: Customizing Menus and Toolbars*

211 *Chapter 18: Capturing a Predefined Command*

223 *Chapter 19: Putting It All Together*

Section 4: Administrative Object API *(not yet available)*

This API gives someone with administrative privileges the ability to make GroupWise system level changes, such as adding and deleting GroupWise users.

Chapter 20: (not yet available)

Chapter 21: (not yet available)

Chapter 22: (not yet available)

Chapter 23: (not yet available)

Chapter 24: (not yet available)

Introducing the Object API

Chapter 1

Section 1: GroupWise Object API

Sometime you may want to access and manipulate the data in a GroupWise database without bringing up the GroupWise client interface. For example, as a GroupWise user may have a need to automate the sending of a message, search for a specific user in an address book, access a certain document, or do any number of other GroupWise related tasks normally accomplished by using the GroupWise client itself.

The GroupWise Object API was developed to fulfill this need. This API allows application developers to see, use, and manipulate the GroupWise information store from outside GroupWise 5.5 or 6.0. This powerful feature opens up the GroupWise system to third party development. Documentation, sample code and downloads for the Object API are available at <http://developer.novell.com/ndk/gwobjapi.htm>.

You will find that the Object API is a carefully organized representation of the GroupWise information store. It consists of many inter-related objects that will sound familiar to GroupWise developers, such as an Account object, a Message object, a Folder object, etc. Each object has methods and properties which describe that object, and which the developer can access in an application.

The following topics are discussed in this chapter.

Contents:

- Getting Started with the Object API
- Your Application is a COM Client
- How to Access the GroupWise Object API
- Late Binding
- Getting an Application Object
- Logging in

Getting Started with the Object API

One of the most frequently asked questions in the Novell Developer forums is “Where can I download the SDK for GroupWise?” The DLLs and EXEs the programmer accesses are already installed as part of the GroupWise client. It often takes several posts back and forth to get across the idea that the Novell developer site contains only the GroupWise sample code and documentation on how to use what is already installed as part of the GroupWise client, at least as far as Object API is concerned.

Your Application is a COM Client

COM stands for “Component Object Model”, and is Microsoft’s answer to the difficulties of more traditional methods of exposing APIs. Before we begin talking about any individual elements of the Object API, we should discuss how COM comes into play with the Object API, and what is required for the Object API to communicate with the underlying GroupWise information store.

As a starting point, it is important to understand that the GroupWise databases are in a proprietary, encrypted, format, and cannot be accessed directly. As a result, the only way to access the GroupWise databases is through the GroupWise client.

A GroupWise client consists of different components. One of these components is the User Interface (UI), which is what the user sees and interacts with when the client is fully launched. The UI does not access the GroupWise databases directly. Instead, the UI communicates with a component of the client called the GroupWise Object Request Broker (GWORB). The GWORB deals with the code that deals with the databases in “direct” mode, or deals with the code that deals with the Post Office Agent in “client-server” mode.

In addition, GroupWise also uses a “COM server” known as the Object API to access the GWORB. While the GroupWise client has its own UI, using the Object API does not require any UI at all. Instead, it can deal directly with the GWORB. As such, an application written to use the GWOAPI is one that is concerned with accessing GroupWise databases, and not with altering or otherwise manipulating the UI. In fact, the only UI in the entire Object API is the password request dialog box that may be programmed to come up as part of the GroupWise login.

Figure 1 shows how the GroupWise UI, the GWORB, and an application work together.

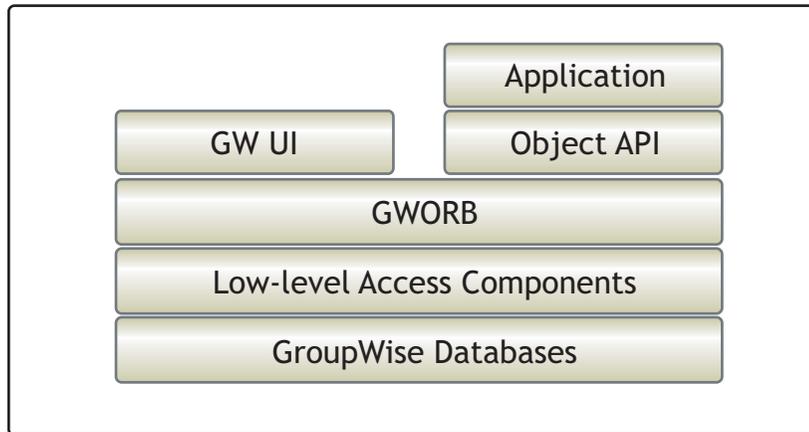


Figure 1: How the GroupWise UI, the GWORB, and an application work together.

What is COM?

COM stands for “Component Object Model”. It is basically a method whereby one piece of software can communicate with and provide services to another piece of software. As far as the Object API is concerned, COM allows GroupWise to provide services to an outside third-party application. It is valuable because it offers a single standard approach for accessing all different kinds of software services (local function calls, system calls, network communications). It also is independent of the programming language being used.

If you are not familiar with the term “COM”, you may know it by one of its other names: *OLE*, *.OCX*, *ActiveX*, or *Automation*. OLE was one of Microsoft’s earlier attempts to solve the dilemma of code control in distributing APIs. It is considered a predecessor to COM. ActiveX is widely regarded simply as another name for COM. For all intents and purposes, ActiveX objects are COM servers, and applications that use them are COM clients.

COM has advantages over traditional APIs for several reasons. One important reason is that traditional APIs are made available to developers as linkable library files, or as well-documented functions kept in a DLL. But these methods of providing APIs pose serious control issues. When a function must be updated for any reason, every developer who has written an application that relies on that function must either re-compile and re-link (if the API is distributed as a linkable library), or they must re-distribute the new DLL (if the API is in a DLL). Even redistributing a DLL may pose additional difficulties not mentioned here.

COM was originally developed to be a platform-independent solution. There were no implementations of COM on anything but Win32 platforms. However, it is now available on a wide range of operating systems from Windows and Windows NT to IBM Mainframes. But by-and-large, it is associated with and used on Win32 platforms.

COM can be a complicated subject to digest. Entire volumes have been dedicated to the subject of how to create simple COM servers (applications that expose some kind of functionality, like an API) and simple COM clients (applications that use functionality or APIs exposed as COM servers). Even more volumes exist to treat the more complex and advance intricacies of COM. Even so, it is a fairly powerful way to expose functionality in an application to other developers. It is well worth learning if you are serious about programming in the Win32 environment. The more you know about COM, the easier it will be to understand the GroupWise Object API.

Where to Get More Information on COM

Wrox Press has published many books by experts in the black arts of COM. If you are not familiar with COM, it is recommended that you read a COM book for beginners. If you do not know C++, it is recommended that you find a book about COM with examples in a language that you do know. For Visual Basic developers, check out *VB COM: Visual Basic 6 Programmer's Introduction to COM*, written by Thomas Lewis and published by Wrox Press in January, 1999. There are several Wrox Press books about COM for C++ developers. If you have access to the World Wide Web, browse to <http://www.wrox.com>. You will see a link to books on COM and COM+, many of which are for C++ developers.

In addition, there are many places on-line to get information about COM. The Microsoft Developer Support Network is a good place to start. You can access this network through Microsoft's main web site at <http://www.microsoft.com>.

Although there are no titles by Wrox Press that treat COM programming specifically for Delphi programmers, we did find one book by Macmillan Technical Publishing entitled *Delphi COM Programming* by Eric Harmon and Al Valvano, published in January, 2000.

Language Considerations

Languages like Visual Basic and Delphi abstract many of the complexities required to write a COM application. In general, C/C++ development environments don't hide the details of COM. Writing COM applications in C/C++ typically requires the developer to know all of the ugly ins-and-outs of COM. (One notable exception, and there may be others, is Borland's C++ Builder. It's VCL framework borrows heavily from the same framework that Delphi uses, making much of the simplifications of writing COM applications in Delphi available to C++ programmers who use C++ Builder.)

If you program in Visual Basic or Delphi, you may be familiar with "Automation Objects" or ActiveX. These are terms that apply to Visual Basic and Delphi support of COM. It is through these mechanisms that your Visual Basic or Delphi applications make use of the GroupWise Object API.

If you peruse the GroupWise API documentation, you may notice that many of the examples in the documentation are in Visual Basic or Delphi. This is because a single line of code in Visual Basic that supports an operation on an Automation Object can sometimes translate into half a dozen lines of code in C++.

Writing COM applications in C++ can be so intricate that it is often difficult to separate the code that specifically supports the use of the GroupWise Object API from the code that is needed to support COM applications in general. This makes it difficult to focus on exactly what the application needs in order to manipulate GroupWise. Realistically, if you want to write COM applications in C++ (including applications that use the Object API, you should engage in a thorough study of the topic of COM.

Languages Used in this Book. Because Visual Basic, Delphi, and C++ are all languages used by GroupWise developers, this book will include examples in all three languages. An emphasis is given to Visual Basic and Delphi, since these are the language most often used by GroupWise developers.

How to Access the GroupWise Object API

COM is a very object-oriented specification. COM servers allow COM clients to create objects in their own memory space that are defined by the COM server itself. For example, the GroupWise client allows your application to create an object known as the “Application Object”. In Object Oriented Programming, (OOP), objects have attributes and methods (sometimes referred to as “properties and methods”, or “data members and functions”). COM objects are no different.

When your application (the COM client) requests that the GroupWise Object API (the COM server) create an Application Object for you, you get an object that has certain properties and methods. The bulk of the GroupWise Object API documentation is a description of these properties and methods.

While all COM servers (such as the GroupWise Object API) create objects in pretty much the same manner, there are differences from one COM server to the next. In addition, the method for getting an object from a COM server varies from one programming language to another.

In the Win32 environment, COM servers are stored in DLLs or in EXEs. When a COM server is stored in a DLL, it is known as an “In-process server” or “InProc Server”. This means that the COM server is loaded by the COM client in the client’s address space. In this manner, the COM client can easily access and control the COM server. COM servers stored in EXEs can be InProc Servers, but are more often known as “OutOfProc Servers” or “Local Servers” because when they are loaded, they usually have their own address space. The client process does not own the server process.

The GroupWise Object API is an InProc server, and is stored in a DLL. However, COM does not require the client application to know which DLL it is stored in. Nor does the client application have to worry about loading the DLL. The Win32 operating system does that for you. If some other COM client wanted to use the services of the GroupWise Object API, the DLL that contains the GWOAPI COM server will already be loaded. Again, this is a detail that the COM client, and you as its developer, do not have to know or worry about.

When your COM client requests the services of a COM server, the request is made by a special ID known as the ProgID (short for Program ID). Each COM server has a unique ProgID. In the case of the GroupWise Object API, the ProgID is “NovellGroupWareSession”. The Windows registry links this ProgID with the actual DLL that contains the COM server’s code. In the case of the GWOAPI, the COM server code is stored in the following DLL: **GWCMA1.DLL**. This file will either be on your local hard drive or on the file server in the GW Client Distribution Directory, depending on whether the client was installed using a “Standard Installation” or a “Workstation Installation”.

But again, you don’t have to load this DLL. When you request the services of the “NovellGroupWareSession” COM server, Windows will load the DLL for you (if it isn’t already loaded). Likewise, you don’t have to worry about unloading it. Windows keeps track of when it should be unloaded.

Late Binding

When you study the topic of COM, you learn that one of its strengths is that the details of COM objects are hidden from the applications that use them. This makes it possible for the designer of a COM object to modify the specification of an object without “breaking” another developer’s application that uses the object.

While this is a very powerful mechanism for separating the details of the object from the application that uses the object, it introduces a lot of necessary overhead in the COM framework. Development languages such as C/C++ and Object Pascal (Delphi) are “strongly typed” languages, which means that the compiler checks assignments and references to make sure that the developer hasn’t made a mistake like trying to assign a “real” (decimal point) value to a character variable.

But COM makes such “type checking” impossible to do at compile time, (at least without a “type library”). Your program won’t know the types of the data members of COM objects it is using until run-time, making it impossible for the compiler to do the checking for you at compile-time. Instead, your program must depend on a COM object to perform its own type-checking at run-time, which is known as “late binding”.

For example, when you develop a COM application, the compiler knows that the COM objects you access have attributes that it doesn't know about. The compiler will, therefore, allow you to perform operations in your program without the strict type-checking it reserves for non-COM applications. It knows that the COM objects will enforce their own type-checking at run-time.

Your application is bound by the type constraints of the object *after* the compilation is done. If your application violates the type checking rules of the COM object, the COM object will gracefully inform your application through an "exception" at run-time.

Early Binding Using Type Libraries

In contrast to the technique of "late binding" with early binding, made possible when you import what is known as a "type library" into your application, your application is bound by the type constraints of the object's properties *before* the compilation is done. The Type Library consists of the definitions of the COM object's data members and members, and is offered by most (not all) COM servers.

Importing Type Libraries. Before a compiler can make use of the GroupWise type library, you must import it. Each compiler has its own method for importing type libraries. Compilers that can create COM applications typically can "import" a type library at design-time. Armed with the information in the type library, the compiler can perform compile-time type checking for your COM application.

The GroupWise Object API provides a type library that is used by a compiler to do type-checking before compilation.

Type libraries are usually stored in the same DLL or EXE where the COM server code is stored. In the case of the GWOAPI, the COM server code is in a DLL file called GWCMA1.DLL. This is also where the GroupWise type library for the GWOAPI is stored.

Type libraries are usually registered with Windows in the Windows registry. This registration will include the name and location of the type library. A compiler that allows you to import a type library will usually present you with a list of type libraries that have been registered with Windows, so you don't really need to know what DLL the type library is stored in. You just need to know its name. In the case of the GWOAPI, the type library is called "GroupWare type library (v1.0)". If you don't see this in the list of available type libraries when you try to import it, you may be able to specify the location of the GWCMA1.DLL file (it will either be on your hard drive, or on a network drive). But be aware that if the GroupWise type library is not in the list of available type libraries when you try to import it, it has not been properly registered with Windows. This means that the GroupWise client most likely has not been installed properly. In this case, it is likely that your application will not be able to successfully use the GWOAPI. You should re-install the GroupWise client.

Reasons For Using Type Libraries. So, why would you be concerned about early-binding vs. late-binding? There are two main reasons for using early-binding:

1. *Reduction of run-time errors.* If you rely on late-binding, and an error in your program is trapped by the COM object you are using, the COM object (hopefully) gracefully handles the error, and typically, no one gets hurt. However, it is still a runtime error, and hurts the image of the application.
2. *Performance.* If you rely on late-binding, there is a lot of overhead introduced in forcing the COM object to perform type checking. It slows your application down quite a bit.

In addition to these two reasons, Visual Basic, through its Object Browser, adds the ability to look at Object syntax as you are creating code. This is very helpful in speeding up the coding process.

Setting up VB

The first thing to do when undertaking a Visual Basic or VBA project involving GroupWise is to enable the **GroupWare type library** in the Project References shown in Figure 2. In VB, follow the menu Project | References and check the GroupWare type library box. In VBA it is Tools | Properties. (It gets moved around a lot. Check the VB or VBA documentation for your specific version).

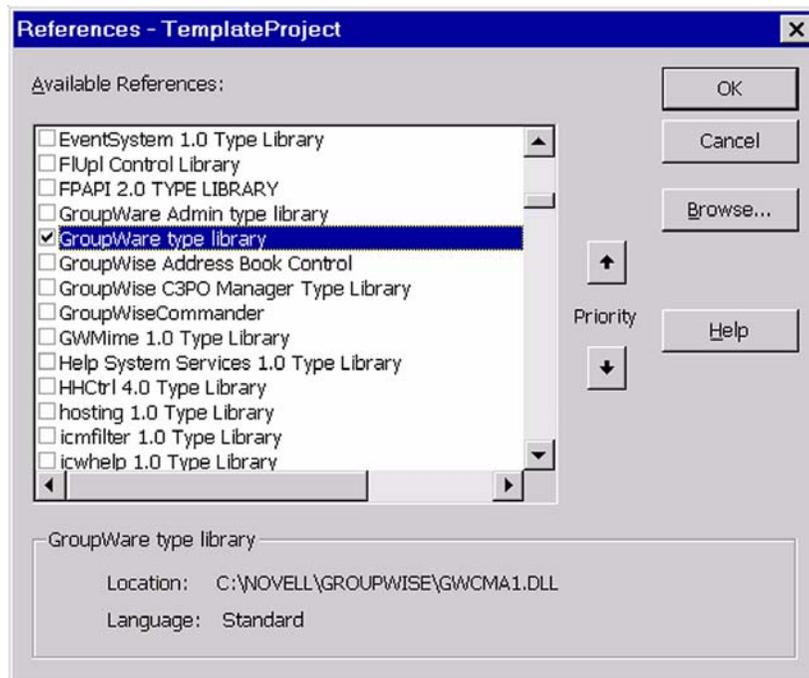


Figure 2: Enable the GroupWare type library in the Project References.

Note: If you can't find GroupWare type library in the list, then GroupWise is not properly installed. You can browse to GWCMA1.DLL to include it in the list to write and compile a program, but it will not execute if not properly installed.

OK, we've said what to do, now here is why— speed. This applies not only because of speed of execution, but speed of programming and speed of debugging. Having a type library is the greatest boon to programming since sliced bread. With the library installed, we can bring up the **Object Browser**, and have a ready searchable reference of all objects, properties, methods, and enumerators of the Object API. These issues are probably more important in VB than any other language.

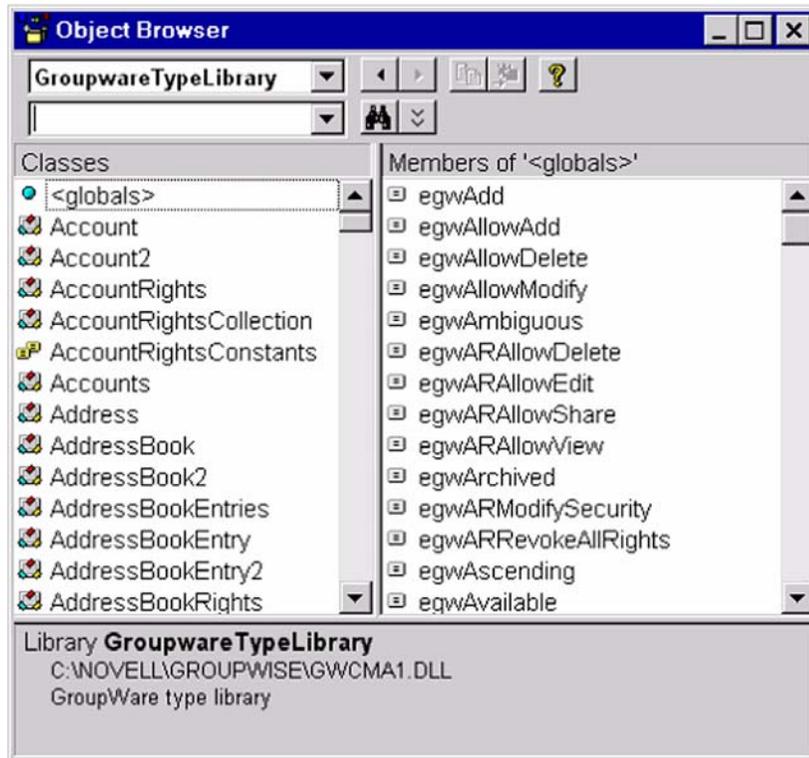


Figure 3: GroupWare type library.

Want to look up the syntax of the Application.Login method? Easy, note that it returns an Account object.

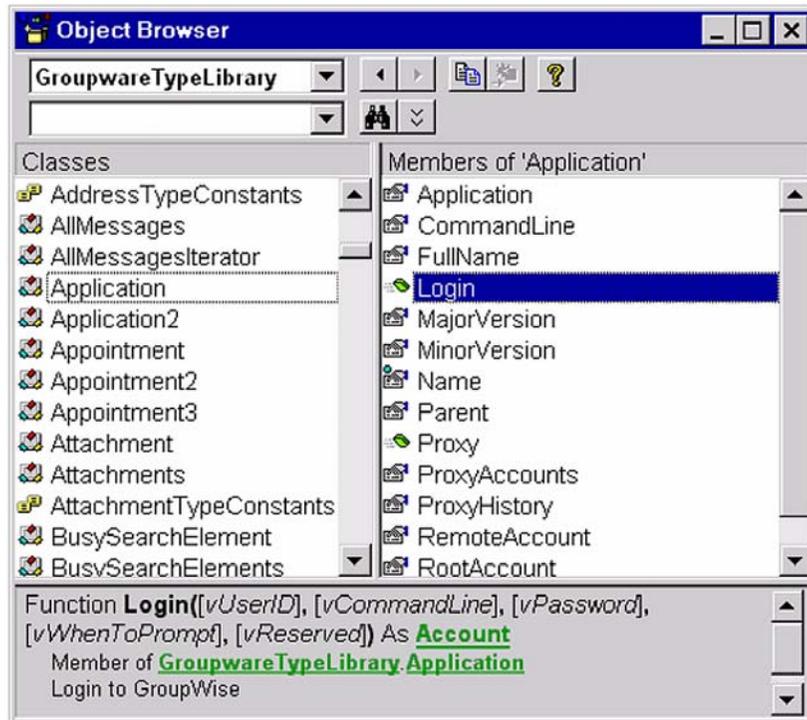


Figure 4: Look up the syntax of the Application.Login method.

Using Early Binding. Once you enable the GroupWare type library, you now can enjoy the benefits of early binding. Having an easy reference is only the beginning.

To understand the benefits of early binding most clearly, first consider what happens when you don't invoke the GroupWare type library and you use late binding instead. You dimension an object as type "Object" or "Variant" as shown below:

```
Dim gwApplication As Object
```

In this case, VB (or VBA) doesn't know how you plan to use gwApplication. It doesn't know when you are editing the code and it doesn't know when it compiles the code. It finally finds out when you assign some object to it at runtime.

This means that VB can't help you select from the known properties and methods while editing or detect type mismatches while compiling. It only detects these problems when run. VB must also take extra time when running to make sure that ogwApp does support the methods we plan to use (or crash with an error message saying something like "Object or With clause does not support method"). Users love this.

Now lets do it the right way:

```
Dim gwApplication As GroupwareTypeLibrary.Application
```

Now VB helps us fill in the details as we type, which makes for faster coding. VB detects more errors while editing and at compile time (rather than at run time), which makes for faster debugging. And VB doesn't have to check while running whether our object supports the method we just called. It knew at compile time, and it runs much faster. That is what is meant by speed!

So why do we write it as GroupwareTypeLibrary.Application? Why not just Application? The answer is that several different type libraries may include an object with the same name. While you can set the sequence used by the compiler to search the type libraries, this is not reliable in the long run. It is generally accepted good VB programming practice to fully qualify the declaration of objects. As Figure 5 shows, there are already two Application objects. One is from GroupWareTypeLibrary, the other is from Word.

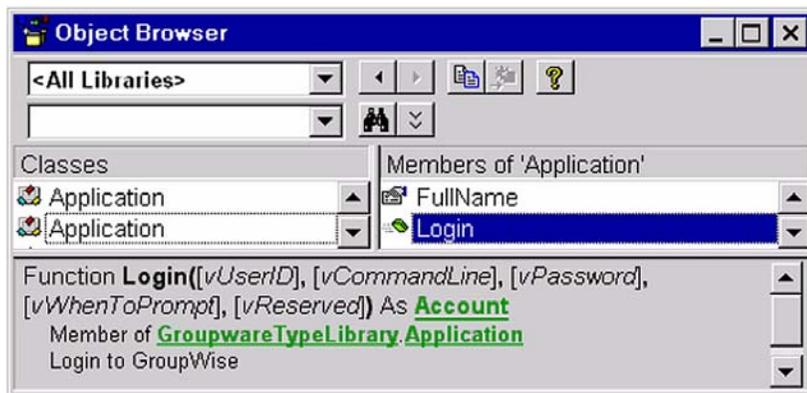


Figure 5: Two Application objects.

Setting up Delphi

If you want to use the GroupWise Type Library in Delphi, select *Project | Import Type Library*, and then select "Groupware Type Library". You can then add the imported type library file to your uses clause and, once you have compiled, you can use *View | Browser* to view the information. The browser in Delphi is not a developed as the one in VB; consequently, you may not receive as much value from it.

Setting up C++

In C++, the normal way to use information in a type library is through the use of an interface called IDispatch. Examples later in this text will show how to use the IDispatch interface to access the various objects in the Object API. These examples will all use early binding, since this is the normal way to access these objects. These examples make use of the “gwoapi.h” header file where object information is stored.

In contrast to early binding, late binding requires more effort to code, a lot more. You have to access an interface to the type library dynamically and use various methods of the type library interface to search for objects or methods. You then must use the “invoke” method of the IDispatch interface to get the desired object. (The GroupWise early binding examples do not use the invoke method.)

Getting an Application Object

Now that we have talked a little about what the Object API is and how COM and type libraries come into play, it is time to show how to actually get started writing an application using the GroupWise Object API.

The top level object in the GroupWise Object API, the object that all other GroupWise objects derive from, is the Application object. So we must begin our application by obtaining a reference to it.

Release 5.5 of GroupWise introduced a new and improved Application object called Application2. (You can see this by examining the Groupware type library). This object includes two new properties that the Application object doesn't have, (MainAccounts and ProxyHistory) and a new method (MultiLogin). If it is available on the version you are using, use it. Its new functionality may be useful to you now or in the future.

Visual Basic

Visual Basic gives us several options to get an initial Application object. We may use either the New keyword or the CreateObject statement. Either method works.

```
Dim gwApplication As GroupwareTypeLibrary.Application2
Set gwApplication = New GroupwareTypeLibrary.Application2
```

Alternately:

```
Set gwApplication = CreateObject("NovellGroupwareSession")
```

That's all there is to it.

Note: The "Set" statement should be used in Visual Basic whenever an object is being referenced on the left side of an assignment statement. It is a reference variable, and does not assign, but rather points gwApplication to the object that the variable represents. It can't "assign" it because a variable can't hold an outside application.

Delphi

Using the Object API in Delphi is also easy. All you need to do is instantiate the object in a variant variable. This will create the Application2 object. Note the slight difference in syntax between VB (CreateObject) and Delphi (CreateOleObject).

```
var gwApplication:variant;  
gwApplication:=CreateOleObject('NovellGroupWareSession');
```

This "application" object can now be used to log into GroupWise accounts, as discussed below.

C++

As mentioned before, C++ is more difficult to use when accessing the GroupWise Object API. Here is some sample code.

```
IGWSession* pIGWSession;  
  
If (FAILED(CoCreateInstance(CLSID_GROUPWISE, NULL, CLSCTX_INPROC_SERVER |  
CLSCTX_INPROC_HANDLER | CLSCTX_LOCAL_SERVER, IID_IGWSession,  
(void*)&pIGWSession))  
    ErrorMessage("Could not create session object");  
    return FALSE;  
}
```

First, an interface pointer to a session (application) object is declared. The interface is defined in gwoapi.h. Next, CoCreateInstance is called. CoCreateInstance is a COM library function that takes a number of arguments. The first argument is the class ID of the object to be created (the Session object). This ID is used to look up an entry in the system registry that maps to a server capable of creating a class object with this type of ID. The third argument specifies what kind of server COM should start. In this example, we will take any of three types. The fourth argument specifies which interface of the object you want (objects can have multiple interfaces). An address to a Session pointer is returned.

Again, this illustrates that if you want to use the Object API with C++, it is extremely helpful to know the basics of COM before you begin.

Now that we have a reference to the Application object, let's work with its most commonly used methods: Login, Proxy, and MultiLogin.

Logging in

Once you have gotten a reference to the GroupWise “Application” object, the next task is to get the GroupWise “Account” object, which represents the desired GroupWise account that is to be accessed. There are three different methods of the application object which return account objects – Login, MultiLogin, and Proxy.

If you are not already accustomed to logging into GroupWise using the Object API, please pay attention to this section. In it you will find code explaining how to provide command line prompts as you log in, including an undocumented command line password switch, and how to avoid the login dialog box. Also this section deals with a problem many programmers encounter – logging into the wrong account.

Let’s begin with the syntax of the `Application.Login()` method. Note that in the GroupWise Object API documentation, all “optional” parameters have brackets around them “[]”. For the login method, all parameters are optional. If successful, the Login method returns an “Account” object.

```
Login([UserID], [CommandLine], [Password], [WhenToPrompt], [Reserved])  
As Account
```

Parameter	Data Type(s)	Description
UserID	(optional) String	UserID of the account to login to. This is Email name for the account (not the Display name). For many systems this will also be the Netware UserID, depending on how administrators setup GroupWise. This parameter is ignored if some application is already running GroupWise, since GroupWise can have only one RootAccount logged in on a single machine. See discussion below.
CommandLine	(optional) String	Command line parameters used with Grpwise.exe. most commonly used to pass the “/pwd=” (password) switch, but may also be used with other valid switches. See discussion below.
Password	(optional) String	Password for the account actually logged into. Since this may not be the account specified by UserID, using this parameter is discouraged. See discussion below.
WhenToPrompt	(optional) Enum	<code>EgwPromptIfNeeded</code> (default) displays a password prompt if needed. This is the only user interface provided by the ObjectAPI. <code>egwNeverPrompt</code> will return an error if the login information can’t be found.
Reserved	(do not use)	Reserved

You may leave the parameters out as well in order to simply logon to the default or currently logged in account.

Example in VB (early binding using Type Library):

```
Dim gwApplication As GroupwareTypeLibrary.Application2
Dim gwAccount As GroupwareTypeLibrary.Account2

Private Sub AccountLogon()
    Set gwApplication = CreateObject("NovellGroupWareSession")
    Set gwAccount = gwApplication.Login
End Sub
```

Example in Delphi:

```
var gwApplication:variant;

procedure AccountLogon;
begin
try

    if varisempty(GroupWare) then begin {allows for multiple procedure calls}
        gwApplication:=CreateOleObject('NovellGroupWareSession');
        gwAccount:= gwApplication.Login;
    end;
except end;
end;
```

Example in C++:

```
IGWSession* pIGWSession;
VARIANT vUserId, vCmdLine, vPassword, vWhenToPrompt, vReserved;

If (FAILED(CoCreateInstance(CLSID_GROUPWISE, NULL, CLSCTX_INPROC_SERVER |
CLSCTX_INPROC_HANDLER | CLSCTX_LOCAL_SERVER, IID_IGWSession,
(void**)&pIGWSession))
    ErrorMessage("Could not create session object");
    return FALSE;
}

VariantInit (&vUserId);
VariantInit (&vCmdLine);
VariantInit (&vPassword);
VariantInit (&vWhenToPrompt);
VariantInit (&vReserved);

V_VT(&vUserId) = VT_EMPTY;
V_VT(&vCmdLine) = VT_EMPTY;
V_VT(&vPassword) = VT_EMPTY;
V_VT(&vWhenToPrompt) = VT_I2;
V_VT(&vReserved) = VT_EMPTY;

V_I2(&vWhenToPrompt) = 0;

if(!SUCCEEDED(pIGWSession->Login(vUserId, vCmdLine, vPassword,
```

```

        vWhenToPrompt, vReserved, &pDIGWAccount))) {
    ErrorMessage("Couldn't login to session");
    return FALSE;
}

```

Note: UserID is ignored if GroupWise, or an application that uses GroupWise (such as Notify) is running. This can lead to unanticipated results for the unwary.

Let's explore the ramifications of the above statement, and provide some work arounds. Say we develop an application that uses the Personnel account. (An account we set up in GroupWise to appear that the message is coming from the Personnel department rather than from a specific user. The Personnel manager happens to be named JoeUser (what a coincidence).

JoeUser is running Notify in the background in order to be notified when he gets mail. Good for him, we don't want him to miss any messages from JanePresident.

JoeUser runs our application, intended to send messages to new hires from the Personnel department. Our code executes the following:

Example in VB:

```

Dim gwApplication As GroupwareTypeLibrary.Application2
Dim gwAccount As GroupwareTypeLibrary.Account2

Private Sub AccountLogon()
    Set gwApplication = CreateObject("NovellGroupwareSession")
    Set gwAccount = gwApplication.Login("Personnel", , "PersonnelPwd",
    egwNeverPrompt)
End Sub

```

Example in Delphi:

```

var gwApplication:variant;
procedure AccountLogon;
begin
    try
        if varisempty(GroupWise) then begin {allows for multiple procedure calls}
            gwApplication:=CreateOleObject('NovellGroupWareSession');
            gwAccount:= gwApplication.Login('Personnel', , 'PersonnelPwd',
            egwNeverPrompt);
        end;
    except end;
end;

```

Example in C++:

```

IGWSession* pIGWSession;
BSTR bMyID, bMyPWD;
char *pMyID, *pMyPWD;
VARIANT vUserId, vCmdLine, vPassword, vWhenToPrompt, vReserved;

```

```

If (FAILED(CoCreateInstance(CLSID_GROUPWISE, NULL,
CLSCTX_INPROC_SERVER | CLSCTX_INPROC_HANDLER CLSCTX_LOCAL_SERVER,
IID_IGWSession, (void*)&pIGWSession)))
    ErrorMessage("Could not create session object");
    return FALSE;
}
VariantInit (&vCmdLine);
VariantInit (&vReserved);

V_VT(&vCmdLine) = VT_EMPTY;
V_VT(&vReserved) = VT_EMPTY;

pMyID = "Personnel";
bMyID = SysAllocString(TO_OLE_STRING(pMyID));
VariantInit (&vUserId);
V_VT(&vUserId) = VT_EMPTY;
V_BSTR(&vUserId) = bMyID;

pMyPWD = "PersonnelPwd";
bMyPWD = SysAllocString(TO_OLE_STRING(pMyPWD));
VariantInit (&vPassword);
V_VT(&vPassword) = VT_EMPTY;
V_BSTR(&vPassword) = bMyPWD;

VariantInit (&vWhenToPrompt);
V_VT(&vWhenToPrompt) = VT_I2;
V_I2(&vWhenToPrompt) = 1; // means egwNeverPrompt

if(!SUCCEEDED(pIGWSession->Login(vUserId, vCmdLine, vPassword,
vWhenToPrompt, vReserved, &pDIGWAccount))) {
    ErrorMessage("Couldn't login to session");
    return FALSE;
}

```

So what do we have? Answer: gwAccount is still JoeUser's account, and not the Personnel account. Why? Notify was running with user JoeUser, so GroupWise ignored the arguments that relate with Personnel, and logged into JoeUser's account instead.

In fact, in general it doesn't matter what is substituted for any of the arguments. The application can list a faulty UserID, use command line switches pointing to a different PO, use a bad password, or use the egwPromptIfNeeded flag. There will be no prompting needed and the account logged into will be JoeUser's account. Simply stated, all the arguments are ignored if JoeUser is already using GroupWise.

We could require JoeUser to end Notify before using our application, but that would be bad, since he may miss the timely message from JanePresident.

On the other hand, if GroupWise is not already up and running, we are in a different ballgame. JoeUser can log into an account specified by the arguments he provides to the login function, or if he specifies no arguments at all, the login method will search for default login information in the operating system or NetWare – such as the UserID and password.

If JoeUser has multiple accounts – say Account A and Account B – then he needs to provide enough information in the Login command to distinguish between the two accounts when he logs in. Which account does he want, and what default login information exists in the system? He may need to provide the specific post office in the second parameter.

Now JoeUser can log into the Personnel account. He must specify the Personnel ID as the first parameter, and PersonnelPwd as the password in the third parameter. If he makes a mistake (like entering a faulty ID, or mismatching the password and ID), and specifies the egwPromptIfNeeded flag, then a popup box will allow JoeUser to reenter the correct information. (This is the only login the Object API provides). If he instead specifies the egwNeverPrompt parameter while making a mistake with the UserID or password, then he will get a runtime error.

JoeUser also has an opportunity to use any of several command line switches in the second parameter. The list of these switches can be found in the GroupWise Client help – under “command line switches”. For example, in Visual Basic, he might try:

```
Set gwAccount = gwApplication.Login("Personnel", "/ipa-144.68.711.119  
/ipp-1677", "PersonnelPwd", egwNeverPrompt)
```

Where the ipa and ipp switches determine the ip address and port.

There is also an undocumented password switch that may come in handy:

```
An undocumented command line switch for starting GrpWise.exe is:  
/pwd=<PASSWORD>  
which provides a password to start GroupWise.
```

Example in VB:

```
Dim gwApplication As GroupwareTypeLibrary.Application2  
Dim gwAccount As GroupwareTypeLibrary.Account2  
  
Private Sub AccountLogon()  
    Set gwApplication = CreateObject("NovellGroupwareSession")  
    Set gwAccount = gwApplication.Login("Personnel", "/pwd=PersonnelPwd",  
    ,egwNeverPrompt)  
End Sub
```

Example in Delphi:

```
var gwApplication:variant;  
procedure AccountLogon;  
begin  
try
```

```

    if varisempty(GroupWise) then begin {allows for multiple procedure calls}
        gwApplication:=CreateOleObject('NovellGroupWareSession');
        gwAccount:=gwApplication.Login('Personnel', '/pwd=PersonnelPwd', ,
egwNeverPrompt);
    end;
except end;
end;

```

The necessity of a password depends upon the default security level involved, and the password options set by each user. “If” a password is required, it can be supplied by the third parameter, or by the /pwd switch in the second parameter.

Finally, if you are not sure you have logged into the right account, you can always check the `Account.Owner.DisplayName` property. If the account name is correct, then all is well. If not, you can `Proxy()` or, if we are using GroupWise 5.5 or later, `MultiLogin()` to get another `Account` object.

Proxy

The `Application.Proxy()` method allows you to proxy to another account with Object API.

```
Application.Proxy(User as {String or Address})
```

Parameter	Data Type(s)	Description
User	String or Address	Either an EmailName string or Address object of the account to proxy.

Let’s look at our personnel problem. If JoeUser logs into his own account, and the other Personnel account has granted JoeUser proxy rights into the Personnel account, then JoeUser can proxy into the Personnel account by using the Proxy method of the Application object.

Example in VB:

```

Dim gwApplication As GroupwareTypeLibrary.Application2
Dim gwAccount As GroupwareTypeLibrary.Account2

Private Sub AccountLogon()
Set GroupWise = CreateObject("NovellGroupwareSession")
Set gwAccount = gwApplication.Login("Personnel", "/pwd=PersonnelPwd",
, egwNeverPrompt)

If gwAccount.Owner.DisplayName <> "Personnel Dept" Then
    Dim gwProxyAcct As GroupwareTypeLibrary.Account2
    Set gwProxyAcct = gwAccount.Proxy("Personnel")

If gwProxyAcct is Nothing Then
    MsgBox "Proxy failed, make sure " & gwAccount.Owner.DisplayName & _
        "has proxy rights to Personnel Dept"
'<bail out code here, such as End ,Exit Sub, Err.Raise(), . . . >

```

```

End If

Set gwAccount = Nothing
Set gwAccount = gwProxyAcct
Set gwProxyAcct = Nothing
End If
End Sub

```

Example in Delphi:

```

var gwApplication:variant; {make this global usually}
procedure AccountLogon;
begin
try
  if varisempty(GroupWise) then begin {allows for multiple procedure calls}
    gwApplication:=CreateOleObject('NovellGroupWareSession');
    gwAccount:= gwApplication.Login('Personnel', '/pwd=PersonnelPwd', ,
egwNeverPrompt);
  end;
except end;
end;

```

Multilogin

GroupWise 5.5 and later provides a better solution for our login problem: the `Application.MultiLogin()` method. Using the previously discussed login method, we can only log into one account at a time. On the other hand, the `MultiLogin` method makes it possible to log into one account, like `JoeUser's`, and then successfully log into another account, such as `Personnel`.

```
MultiLogin(UserID, [CommandLine], [Password], [WhenToPrompt], [Reserved]) As Account
```

Parameter	Data Type(s)	Description
UserID	String	UserID of the account to login to. This is Email name for the account (not the Display name). For many systems this will also be the Netware UserID, depending on how administrators setup GroupWise.
CommandLine	(optional) String	Command line parameters (switches) used with Grpwise.exe.
Password	(optional) String	Password for the account specified by UserID.
WhenToPrompt	(optional) Enum	egwNeverPrompt (default), the login method returns Nothing if the login is unsuccessful egwPromptIfNeeded displays a password prompt if needed. This is the only user interface provided by the ObjectAPI.
Reserved	(do not use)	Reserved

Note: The `MultiLogin` method always returns the specified account or `Nothing` (if the account can't be returned).

This method makes life a lot easier. If we got the account UserID (and Password, if needed) correct, then we get the account we specify without question. Note these differences:

Feature	Login	MultiLogin
Adds an account to Application.MainAccounts collection	Yes	Yes
Sets Application.RootAccount property	Yes	No
Preferred password passing parameter	2nd (CommandLine) with "/pwd=" prefix	3rd (Password)
Default WhenToPrompt parameter	egwPromptIfNeeded	egwNeverPrompt
The UserID parameter	Is optional	Is required
The Account returned (if it is possible to return the account)	The account specified by the UserID parameter is overwritten if GroupWise is already running by the account which is logged in. GroupWise is running if any application based on it (i.e.Notify, InForms, WorkFlow) is running.	The account specified by UserID.
GroupWise version	All versions with Object API	GroupWise 5.5 and later

There are a couple of “gotchas” associated with using the MultiLogin command.

First, suppose you were to try and open five user accounts, with UserIDs identified by UserName(1-5) and passwords identified by UserPassWord(1-5). Suppose your Visual Basic code went something like this:

```
For i = 1 To 5
    Set gwAccount = gwApplication.MultiLogin(UserName(i), , "UserPassWord(i))
Next
```

If you were to try this, you would find that the number of total accounts you have opened is only 1 — not 5. This is because you are using the same gwAccount object during every iteration. This account object is being overwritten, and during loops 2-5 the previous gwAccount object is probably being released by Visual Basic.

To correct this problem, make sure that you use a distinct name for each Account you open.

A second “gotcha” is trying to use the GroupWise Personal Address Books collection derived from one of the multiple accounts you may open. Normally, after accessing a GroupWise account object, you can drill down and get the address books used by that particular account.

The problem is that the GroupWise Object API uses MAPI to talk to any address book. The MAPI system, however, logs into one user only – the same user that a call to the `gwApplication.Login` would return. Thus, only the address book collection used by the account returned by `gwApplication.Login` is correct. When MultiLogin is used to log into any other person’s account, the address books and their associated entries will be incorrect and will come from the `gwApplication.Login` account instead.

Security and Maintenance Considerations for Login

When developing an application, one of the questions you may deal with is how to store the password parameter(s) used. Storing them in the application is convenient, but is usually in human readable form leading to a security risk. There are alternative strategies to deal with this.

Option	Security	Maintenance
Store password for the application account locally	Security hole for one account	Synchronize password changes or set password to never change
Allow selected users to proxy the account, Use <code>Login()</code> without parameters for user to login to their own account (if not already running), then always Proxy	Security is maintained	Updating proxy rights as users change

Application Object Properties

Besides the three login methods of the Application object that we have discussed, there are also several Application object “properties”. They are all “read only”. A description of each of these properties is listed in the GroupWise documentation found at the GroupWise Object API developer site <http://developer.novell.com/ndk/gwobjapi.htm>. Properties which are also objects will be discussed in the following chapters.

Summary

In this chapter we learned that the Application object is the root object used by the GroupWise object ABI. In the next chapter we will discuss using the Account object.

Copyright © 2002 by Novell, Inc. All rights reserved.
 No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, for any purpose without the express written permission of Novell.

All product names mentioned are trademarks of their respective companies or distributors.

Using the Account Object

Chapter 2

Section 1: GroupWise Object API

In Chapter 1, we discussed the GroupWise Application object. We learned that the Application object is the root object used by the GroupWise object API. All other GroupWise objects derive from the Application object. We also learned that the Application object has three different methods available, and that each of these methods returns a GroupWise Account object. These methods are:

- Login – used to log into a user’s root account.
- Proxy – used to proxy into someone else’s account who has granted you proxy rights.
- MultiLogin – used to log into multiple accounts simultaneously.

All of the above methods return an Account object. Because of the importance of the Account object, this chapter will concentrate on describing some of its methods and properties. This description includes several new properties and methods that have been made available for GroupWise 6.0 SP1 and GroupWise 5.5 EP SP4. In terms of the number of methods and properties available, the Account Object is one of the larger objects in the Object API.

The following topics are discussed in this chapter.

Contents:

- AccountRights Property
- ProxyAccounts and Proxied Properties
- DefaultAccountRights Property
- AddressBook, Folder, and Message Properties
- Archive Properties
- Other Properties
- Account Object Methods

AccountRights Property

One of the important properties of the Account object is the AccountRights property, which is an object of type “AccountRightsCollection”. This property represents proxy access rights that a given user grants to other users.

For example, suppose User1 grants proxy rights in his account to User2, User3, and User4. If User1 then uses the object API to access the AccountRights property of his own Account object, this property will be an AccountRightsCollection object consisting of 3 “AccountRights” objects in the collection, corresponding to the proxy rights granted to User2, User3, and User4 respectively. (These AccountRights objects are not to be confused with the AccountRights property of the Account object.) By accessing each object in this collection, and using its properties and methods, User1 can get information about the proxy rights he granted to each of the other 3 users.

In contrast, suppose User2 proxied into User1’s account and looked at this same collection. What would User2 see?

This time User2 would only see 1 item in the collection – not 3. The only AccountRights object he would see would be the rights that User1 granted to User2. The GroupWise architects did it this way so that User2 would not see all the users to whom User1 granted proxy rights. This should be User1’s private information.

AccountRightsCollection

Properties. The AccountRightsCollection object consists of the usual properties for a collection. The most often used property is the “Count” property, which gives the number of AccountRights objects in the collection. It can be used to iterate through the list of AccountRights objects.

Methods. There are two methods of the AccountRightsCollection object.

1. Use the `Item()` method to obtain a specific AccountRights object in the collection.

`Item(Index)`

`Item()` takes as single parameter called `Index`, which is of type “Long”. You can make `Index` an integer between 1 and `Messages.Count`, and the `Message` object corresponding to that position will be retrieved. This is useful when iterating within a loop.

2. Use the `Add()` method to add a new AccountRights object to the collection. This translates to giving some new user proxy rights into your account.

```
AccountRightsCollection.Add(Address, RightsBitMask)
```

Name	Data Type	Description
Address	Object	This parameter can be an Address Object, or a derivative of the Address object such as an AddressBookEntry object (see Chapter 7: <i>Understanding Address and AddressBook Objects</i>). It can even be a AddressBookCollection object. It therefore is used to tell what user or users should have proxy rights into the account.
RightsBitMask	Long	Specifies what account rights are to be given. It is derived by using a bitwise inclusive OR operator to the AccountRightConstants given in the Object API documentation

For example, to give proxy rights to some person listed in the GroupWise address book, (assuming you have found the particular AddressBookEntry object associated with that entry - see Chapter 7: *Understanding Address and AddressBook Objects*), you could do the following:

```
Set MyAccRights = MyAccount.AccountRights  
MyAccRights.Add(UserEntry, 12)
```

By consulting the AccountRightsConstants in the GroupWise documentation, you can determine that the value of egwReadMailAndPhone is 8, and the value of egwReadAppointments is 4. Using the “OR” operator, a value of 12 would give the user the rights to read mail messages, phone messages, and appointments.

AccountRights

Properties. Most of the properties of the AccountRights object are simply read/write boolean values indicating whether or not a “particular” right has been granted – such as AccountRights.ReadAppointments. If this value is true, the particular user associated with the AccountRights object can read appointments. If it is false, he cannot.

The user associated with the AccountRights object is given by the “Address” property, which is read only.

There is also a BitMask property which gives the rights described in the last section. This can be modified if desired using the AccountRightsConstants and the bitwise inclusive OR operator.

Methods. The only method available is the Delete method. It will delete that particular user from the AccountRightsCollection, causing the AccountRightsCollection to be reduced by one.

ProxyAccounts and Proxied Properties

ProxyAccounts. Closely related to the AccountRights property is the ProxyAccounts property, but they should not be confused. As discussed earlier, through the AccountRights property User1 can discover all the users to whom he has granted proxy rights, and what rights each user has. It is a collection of “AccountRights” objects. On the other hand, the ProxyAccounts collection lists all the proxy accounts that the owner of the account is currently using himself – and has nothing to do with the proxy rights he has granted to others. It is a collection of “Accounts”.

Using the same example described earlier, if User1 grants proxy rights to User2, User3, and User4; User1’s AccountRights collection will consist of 3 elements. Each element will be an AccountRights object that will describe to whom User1 gave proxy rights and what those proxy rights are. If User2 is currently taking advantage of this proxy right granted by User1 and is proxying into User1’s account, then User2’s ProxyAccounts collection will have one element in it – consisting of User1’s account. If User2 is proxying into someone else’s account as well, then the ProxyAccounts collection would consist of two elements. If he is not proxying into anyone’s account, the collection would be empty.

Proxied. The Proxied property is a read only boolean value which simply indicates whether an active account has been proxied into or not. If User2 has proxied into User1’s account and tests this value, it will be true. If he logs into his own account, its value will be false.

DefaultAccountRights Property

The DefaultAccountRights property lists the access rights given to all users not listed in the AccountRights property. These are common rights shared among all users, to whom the account owner has not specifically granted rights. This property is “read only”, and is set through the GroupWise client (see Tools | Options | Security Icon | <All User Access>). It is only accessible from the root account.

Note that this property is not a collection, but consists of one “AccountRights” object that describes these common rights. The account owner should be careful and make sure that he has not granted general access rights into his account through the client. This has been done before by accident.

AddressBook, Folder, and Message Properties

The Account object includes properties that allow the user to access collections of address books, folders, messages, document libraries, filters, and field definitions as well as access to individual address books and folders. Since these object types are discussed in other chapters, a thorough description of these types of objects is best left to these other chapters. For purposes of this chapter, we will simply mention what these properties are.

Collection Type Properties

The following are collection type properties.

AddressBooks. The AddressBooks property of the Account object is an object of type “AddressBooks”. This property represents a collection of all the address books associated with the root account. They may be both system address books and personal address books. See Chapter 7: Understanding Address and AddressBook Objects.

AllFolders. The AllFolders property is a “Folders” collection object, representing all the folders owned by this account. It includes all shared folders, even if they are owned by another user. See Chapter 3: Understanding Folder and Trash Related Objects.

AllMessages. The AllMessages property is a “AllMessages” collection object, representing all the messages in the account. It does not include messages from other accounts even if they appear in this accounts query folders and shared folders. See Chapter 4: *Message Collections*.

DocumentLibraries. The DocumentLibraries property is a “DocumentLibraries” collection object that represents all the document libraries seen by the account. See Chapter 6: Understanding Document and Document Related Objects.

Filters. The Filters property is a “Filters” collection object that represents all the saved filters of the account. See Chapter 9: Understanding Filter and Query Related Objects.

FieldDefinitions. The FieldDefinitions property is a “FieldDefinitions” collection object. It represents all the field definitions that can be associated with the account. See Chapter 8: *Understanding Field and Field Related Objects*.

Non-Collection Type Properties

The following are non-collection type properties.

Cabinet, Calendar, Mailbox, RootFolder, WorkFolder, DocumentsFolder. These individual folders can each be accessed directly from the Account object. See Chapter 3: *Understanding Folder and Trash Related Objects*.

DefaultAddressBook, SystemAddressBook, FrequentContacts. These individual address books can each be accessed directly from the Account object. See Chapter 7: *Understanding Address and AddressBook Objects*.

DefaultDocumentLibrary. The default document library can be accessed directly from the Account object. See Chapter 6: *Understanding Document and Document Related Objects*.

Archive Properties

There are several archive properties associated with the Account object. These properties are listed below:

Archived. This read only boolean property describes whether a given account is an archive account or not. It is available for GroupWise versions 5.5 and later.

DefaultPathToArchive. This string property describes the path to the current “default” archive account. It is a read only property, so it cannot be set. It is also not affected by using the SetArchiveTo method described later. It can be set in the GroupWise client under the Tools | Options | Environment tab | File Location tab.

PathToArchive. This string property describes the path to the current archive account. Like the DefaultPathToArchive object, it is also read only. However, it can be set by using the SetArchiveTo method of the Account object.

Owner Property

The owner of an account can be found by using the owner property. Since the owner property is of type “Address”, which in turn has a property called “DisplayName”, these properties can be combined to show the DisplayName of an account. Here is how it could easily be done in Visual Basic, with AccountOwner of type String.

```
AccountOwner = MyAccount.Owner.DisplayName
```

Other Properties

A brief summary of a few other Account properties are listed below:

ObjType. The ObjType property is an enumerated value that represents the type of Account object, such as User, Resource, etc.

PathToHost. This string property represents the path-to-host of the GroupWise server that was logged into, or an empty string if the user logged in through TCP/IP.

Remote. This boolean read only property shows whether the account is remote.

TCPIPAddress. This string property shows the TCP/IP address of the GroupWise server that was logged into, or an empty string if the user logged in through a mapped path or a Universal Naming Convention (UNC) path.

TCPIPPort. This Long property gives the TCP/IP port of the GroupWise server that was logged into, or an empty string if the user logged in through a mapped path or a UNC path.

Trash. The Trash property is an object of type Trash representing the trash for this account. The Trash object can be emptied or refreshed, and holds a TrashEntries collection, consisting of the individual items in the trash. The Trash Object is discussed in the Folders chapter.

Account Object Methods

There are 12 methods available from the Account Object. These methods will now be briefly discussed.

Archive Methods

GetArchiveAccount ([VARIANT path]). This method returns the archive Account object specified by an optional argument path. If the argument does not appear, it returns the archive Account object that has been set by SetArchiveTo, or, if SetArchiveTo has not been used, it looks up the Archive account in the database. It is only available from the root account.

SetArchiveTo([path]). This method changes the archive path to the path specified in the argument. If no path is specified, it will change the archive path to the archive path specified in your GroupWise preferences. It is only available from the root account.

MergeArchive (DestinationArchiveAcct). This method merges all messages from the current archive account into the archive account specified by DestinationArchiveAcct.

Here is a Visual Basic example that uses all three of these methods. It uses two archive accounts called MyArchive1 and MyArchive2, stored in C:\archived1 and C:\archived2 respectively. Assume the default archive is MyArchive1. The example also uses some of the account object archive properties discussed earlier.

```
` Set the path to the MyArchive1 account
MyAccount.SetArchiveTo ("C:\archived1")

` Both of these paths will show C:\archived1
MyPath = MyAccount.DefaultPathToArchive
MyPath = MyAccount.PathToArchive

` Get the first archive account object
Set MyArchive1 = MyAccount.GetArchiveAccount()

` Now set the path to C:\archived2 account
MyAccount.SetArchiveTo ("C:\archived2")

` The first path will still show the path to the MyArchive1 account. SetArchiveTo
` does not change the default path. The second will show the path to the
` MyArchive2 account.
MyPath = MyAccount.DefaultPathToArchive
MyPath = MyAccount.PathToArchive

` Get the second archive account
Set MyArchive2 = MyAccount.GetArchiveAccount()

` Merge the MyArchive1 account into the MyArchive2 account
MyArchive1.MergeArchive (MyArchive2)
```

Other Methods

CreateQuery (). This Account object method creates and returns a Query object. The Query object is explained later in Chapter 9: *Understanding Filter and Query Related Objects*. It is helpful when conducting searches.

GetFolder (FolderID). This method returns the folder in this account with the specified ID. The ID is specified as a string. See Chapter 3: *Understanding Folder and Trash Related Objects*.

GetMessage (MessageID). Similar to the GetFolder method, this method returns the message in this account with the specified ID. The ID is specified as a string. See Chapter 4: *Message Collections*.

Refresh (). Calling this method forces the Account object, and all its objects and collections, to reread property values from the message database.

Proxy (User). This method returns the proxy account specified by the User argument. The user argument can be specified as a UserID string of the desired proxy account, or as an address of the desired proxy account.

ConvertEmailAddress (OldAddress, [Format]). This method takes an address, specified as either a string or an Address object, and converts it into a new format specified by Format. It returns the address string with the new format. The format is an enumerated value specified by the EmailAddressFormatConstants. Both the constants and the method are new to GroupWise 5.5.

Some Newer Methods

For GroupWise 6.0 SP1, and GroupWise 5.5 EP SP4 and later versions, there are three new methods available at customer request. Here is a description of these new methods.

SetPassword (OldPassword, NewPassword). Many GroupWise developers have wanted for some time to be able to change their own password by using the Object API. Up until now, this has been possible through the GroupWise Client, but not through the Object API. This new method fulfills this need.

The first argument to SetPassword is simply the old password specified as a string. The second argument is the desired new password, also specified as a string. If the old password is incorrect, an error will result. Hence, you must know your old password to change it. If the GroupWise account had no password, then the OldPassword should be an empty string. Otherwise an error will occur.

Note that this method does not change the user's NetWare, Windows, LDAP or other passwords. It changes only the password associated with the current GroupWise database.

```
SetPassword ("OldPsWrd", "NewPsWrd")
```

SynchronizeToRemote (path, masterPassword, isCache, flags). This new method performs the equivalent of the regular client's "Hit the Road" functionality. Here is a description of the different arguments.

Name	Data Type	Description
path	String	This is the desired name for the remote mailbox or the client cache mailbox.
MasterPassword	String	The user's main mailbox password.
isCache	Boolean	Indicates if the call is setting up regular remote or client cache.
flags	Enumerated Integer	The flags indicate what is to be downloaded. (See the new SynchronizeConstants.)

SynchronizeWithMaster (flags). This new method synchronizes the user's master mailbox in remote mode. The user needs to have set up the connection to the master post office from the regular client or by using the new SynchronizeToRemote above.

Name	Data Type	Description
flags	Enumerated Integer	The flags indicate what is to be downloaded. (See the new SynchronizeConstants.)

Summary

In this chapter we have described several new Account object properties and methods made available for GroupWise 6.0 SP1 and GroupWise 5.5 EP SP4. Chapter 3 will discuss Folder and Trash related objects.

For more information on Accounts, please check the GroupWise Object API documentation <http://developer.novell.com/ndk/doc.htm>, sample code <http://developer.novell.com/ndk/sample.htm> or visit the Developer Support Forum area at <http://developer-forums.novell.com/category/index.tpt>.

Copyright © 2002 by Novell, Inc. All rights reserved.
No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, for any purpose without the express written permission of Novell.

All product names mentioned are trademarks of their respective companies or distributors.

Understanding Folder and Trash Related Objects

Chapter 3

Section 1: GroupWise Object API

Folder objects in GroupWise store messages and folders. The rights of a folder can be determined through the FolderRights object of an associated FolderRightsCollection.

Like a GroupWise folder, the Trash object is used to hold messages or TrashEntries which are likewise accessed through a TrashEntries collection.

The following topics are discussed in this chapter.

Contents:

- Folders
- Accessing Nested Folders
- The Different Types of Folder Objects
- Creating a Folder
- Getting at the Contents of a Folder
- Shared Folders
- Testing a Folder for Shared Status
- Testing a Folder For Folder Rights
- Sharing a Folder
- Accepting a Shared Folder
- Removing User Access to a Shared Folder
- Summary

Folders

Folders, like many other collections objects, have a `Count` property and `Item()` method that allows you to access individual Folder objects. Simply call the `Item()` method with a single variant parameter called `Index`.

```
Folder.Item(Index as Variant)
```

If `Index` is an integer between 1 and `Folders.Count` the method will return the folder corresponding to index as it appears in the GroupWise client. Be sure to check your range to avoid a program error. Use `Folders.Count`, which returns a long integer, to determine how many folders are within a Folders collection.

This is a particularly useful method if you wish to perform operations on all folders within a for loop. The following example iterates through all folders in the cabinet and displays the name of each one in turn:

Example in VB:

```
'gwFolder is a valid reference to the Cabinet object

Dim gwFolders As GroupwareTypeLibrary.Folders
Dim gwFolder As GroupwareTypeLibrary.Folder2

Private Sub WalkFolders(gwFolder)
    Set gwFolders = gwFolder.Folders
    If gwFolders.Count > 0 Then
        For i = 1 to gwFolders.Count
            Set gwFolder = gwFolders.Item(i)
            msgbox gwFolder.Name
        Next i
    End If
End Sub
```

Example in Delphi:

```
//gwFolder is a valid reference to the Cabinet object

procedure WalkFolders (gwFolder:Variant);
var i:integer;gwFolders,gwFolder:variant;
begin
    gwFolders:=gwFolder.Folders;
    if gwFolders.Count > 0 then begin
        For i:=1 to gwFolders.Count do begin
            gwFolder := gwFolders.Item(i);
            ShowMessage(gwFolder.Name);
        end;
    end;
end;
```

Although this method can quickly run through an entire `Folders` collection, it leaves much to be desired, if you wish to find a particular folder, in part because you (and your code) may not know how the end user has ordered the folders in his or her account. If `Index` is a string, it represents a `FolderID` in which case `Folders.Item(Index)` will return the folder within the collection that has a matching `FolderID`. If the `FolderID` is invalid or refers to a folder that is not within the particular `Folders` collection, then the method will throw an exception. This too, leaves much to be desired, as a particular `FolderID` may be unknown. The following method accepts a folder name.

```
Folders.ItemByName(Name as String)
```

This is a tricky method; it will return the first instance of any given folder that matches `Folder.Name`. If, for example, you have several sub-folders with the same name but different parent folders, you should always check the `Name` property of `Folder.ParentFolder` to determine exactly which folder you have obtained.

There is no way to make successive `ItemByName()` calls; thus, you may need to use the `Item()` method and iterate through all of the folders in order to find the folder you are looking for. Because `ItemByName()` may be called by any given `Folders` object, you can limit the folders in which you search. For example, you might search in the `Cabinet`, or you might iterate through each top-level folder in the cabinet using the `Item()` method and then call `ItemByName()` within each top-level folder's collection. The following is an example procedure that searches each top-level folder for the first folder named 'MainFolder' in each folder within the `Cabinet` (but it does not find folders in the `Cabinet` that happen to be called 'MainFolder'):

Example in VB:

```
'gwAccount is a valid Account object

Dim gwFolders As GroupwareTypeLibrary.Folders
Dim gwFolder As GroupwareTypeLibrary.Folder2
Dim gwCurrentFolder As GroupwareTypeLibrary.Folder2

Private Sub FindFoldersByName()
'Get the most recent Folders list.
gwAccount.Refresh

Set gwFolders = gwAccount.Cabinet.Folders
If gwFolders.Count > 0 Then
  For i = 1 To gwFolders.Count
    Set gwCurrentFolder = gwFolders.Item(i)
    On Error GoTo FldrErr
    Set gwFolder = gwCurrentFolder.Folders.ItemByName("MainFolder")
  Next i
End If
```

```
Exit Sub
FldrErr:
    MsgBox "Folder not found: " & Err.Description & Err.Number
End Sub
```

Example in Delphi:

```
//gwAccount is a valid Account object

procedure FindFoldersByName;
var i:integer;gwFolders,gwFolder,gwCurrentFolder:variant;
MainFolder:string;
begin
    gwAccount.Refresh; {This is to get the most recent Folders list}
    MainFolder:='MainFolder';
    gwFolders:=gwAccount.Cabinet.Folders;
    if gwFolders.Count > 0 then begin
        For i:=1 to gwFolders.Count do begin
            gwCurrentFolder := gwFolders.Item(i);
            try
                gwFolder:=gwCurrentFolder.Folders.ItemByName(MainFolder);
                ShowMessage('Found one in folder: '+gwCurrentFolder.Name);
            except
                ShowMessage('No such folder in folder: '+gwCurrentFolder.Name);
            end;
        end;
    end;
end;
```

Accessing Nested Folders

Every Folder object has a Folders property that holds nested folders if they exist. You can use recursion to walk folders more efficiently. Here is a new declaration for WalkFolders that takes advantage of recursion.

Example in VB:

This won't work on folders with no sub folders or second folders.

```
'gwFolder is a valid Folder object

Dim gwFolders As GroupwareTypeLibrary.Folders
Dim gwFolder As GroupwareTypeLibrary.Folder2
Dim gwTempFolder As GroupwareTypeLibrary.Folder2

Private Sub WalkFolders(gwFolder)
Set gwFolders = gwFolder.Folders
If gwFolders.Count > 0 Then
    For i = 1 To gwFolders.Count
        Set gwTempFolder = gwFolders.Item(i)
        MsgBox gwTempFolder.Name
    Next i
End Sub
```

```
        Call WalkFolders(gwTempFolder)
    Next i
End If
End Sub
```

Example in Delphi:

```
//gwFolder is a valid Folder object

procedure WalkFolders (gwFolder:Variant);
var i:integer;gwFolders,gwTempFolder:variant;
begin
    gwFolders:=gwFolder.Folders;
    if gwFolders.Count > 0 then begin
        For i:=1 to gwFolders.Count do begin
            gwTempFolder := gwFolders.Item(i);
            ShowMessage(gwTempFolder.Name);
            WalkFolders(gwTempFolder);
        end;
    end;
end
```

Now if you call `WalkFolders()`, each folder in the Cabinet, and then each sub-folder, all the way to the bottom level will be displayed in sequence. Of course, you will likely never just show the folder names in this manner, but this example is illustrative of the way to recreate a folder structure from GroupWise.

Nesting may create problems with the `Folders.ItemByName()` method because it will search all folders within the given `Folders` collection. Thus, as discussed above, you should examine the `Folder.ParentFolder` property to determine what level of nesting the folder object is in.

The following properties also return a Folder object:

- `Account.Cabinet`: Returns the Cabinet folder, which is a system folder.
- `Account.Calendar`: Returns the Calendar folder, which is a system folder that holds appointment, note, and task items.
- `Account.DocumentsFolder`: Returns the Documents folder, which holds recently used document references.
- `Account.Mailbox`: Returns the Mailbox folder, which is the default delivery location for new mail items.
- `Account.RootFolder`: Returns the top-level folder in the account. By default, this folder holds all other folders including the Mailbox, Calendar, Cabinet, WorkFolder, and DocumentsFolder. The end user can move them in the client, but usually does not do so.

- `Account.WorkFolder`: Returns the Work-In-Progress folder, which holds draft mail messages.
- `Account.GetFolder(FolderID)`: This is actually a method call of the `Account` object. If you happen to know the `FolderID` of a folder (which is stored in the `Folder.FolderID` property), then you can obtain any `Folder` object reference directly with this method.
- `Folder.ParentFolder`: Returns the folder that holds the current `Folder` object. Be careful. If this is the `RootFolder` (the very first folder in the whole system), instead of a folder object, you'll get 'Nothing' which can lead to bugs in your code if not handled.

The Folder Object

The key properties of the `Folder` object are:

Property	Data Type	Description
Description	String (R/W)	Describes the folder.
FolderID	String (R/O)	A unique ID (similar to <code>MessageID</code>) for the given folder. You can use this ID in the <code>Account.GetFolder()</code> method to obtain a reference to any <code>Folder</code> object directly.
Name	String (R/W)	Name of the folder.
ObjType	Enum (R/O)	<code>EgwUnknownFolder</code> , system folders (<code>egwMailbox</code> , <code>egwRoot</code> , <code>egwCabinet</code> , <code>egwCalendar</code> , <code>egwWork</code> , <code>egwDocuments</code>), query folders (<code>egwQuery</code>), and personal folders (<code>egwPersonalFolder</code>). Note that incoming shared folders are considered personal folders in this property.
System	Boolean (R/O)	This property is <code>TRUE</code> if the folder is a system folder.

The following example demonstrates the use of the `Name` and `FolderID`, as well `Account.GetFolder(FolderID)` to find a specific folder in `GroupWise`.

Example in VB:

```
'gwAccount is a valid Account object

Dim gwFolders As GroupwareTypeLibrary.Folders
Dim gwCabinet As GroupwareTypeLibrary.Folder2
Dim gwTempFolder As GroupwareTypeLibrary.Folder2

Private Sub CompareFolder()
'Get the most recent Folders list.
gwAccount.Refresh
```

```
Set gwCabinet = gwAccount.Cabinet
FolderID = gwCabinet.FolderID
Set gwTempFolder = gwAccount.GetFolder(FolderID)

If gwCabinet.Name = gwTempFolder.Name Then
    MsgBox "FolderID worked"
Else
    MsgBox "FolderID failed"
End If
End Sub
```

Example in Delphi:

```
//gwAccount is a valid Account object

procedure CompareFolders;
var gwCabinet,gwTempFolder:variant;
    gwFolderID:string;
begin
    gwCabinet:=gwAccount.Cabinet;
    gwFolderID:=gwCabinet.FolderID;
    gwTempFolder:=gwAccount.GetFolder(gwFolderID);
    if comparetext(gwCabinet.Name,gwTempFolder.Name) then
        ShowMessage('FolderID worked') else
        ShowMessage('FolderID failed');
end;
```

The Different Types of Folder Objects

GroupWise recognizes the following folder types:

Personal

These are general-purpose folders that hold messages or other folders. Users may share their personal folders with other users.

Query

This type of folder is dynamic. It holds only those items that match Query.Expression. For example, the system defined “Sent Items” query folder will execute a query to find all outgoing items, regardless of folder location while the “Task List” query folder will execute a query to find all Tasks associated with the account.

Shared

These are personal folders that provide shared level access to the folder and its contents to other GroupWise users.

System

These are folders defined by the GroupWise client. They cannot be deleted and include the Mailbox folder where new items are deposited; the Documents folder where references to recently opened documents are stored; the Work In Progress folder where draft messages are stored; the Calendar folder where all accepted calendar items reside; and the Trash folder where all deleted messages are kept.

For most operations, folder objects may be treated alike. A few differences are discussed later in this chapter.

Creating a Folder

To create a new folder, use `Folders.Add()`. The following methods will create a new Folder:

```
Folders.Add(Name as String)
```

This method creates a new personal Folder and adds it to the Folders collection with `Folder.Name` set to the string parameter passed to the method and `Folder.ObjType` set to `egwPersonalFolder`.

The following example adds a new personal folder to the Cabinet called “MainFolder”:

Example in VB:

```
'gwAccount is a valid Account object

Dim gwCabinet As GroupwareTypeLibrary.Folder2

Private Sub AddFolder()
Set gwCabinet = gwAccount.Cabinet
On Error GoTo FldrErr
    gwCabinet.Folders.Add "MainFolder"

Exit Sub
FldrErr:
    MsgBox "Error Adding Folder: " & Err.Description & Err.Number
End Sub
```

Example in Delphi:

```
//gwAccount is a valid Account object

procedure AddFolder;
var gwCabinet:variant;MainFolder:string;
begin
    MainFolder:='MainFolder';
```

```
gwCabinet:=gwAccount.Cabinet;  
try  
  gwCabinet.Folders.Add(MainFolder);  
except  
  ShowMessage('Error Adding Folder');  
end;  
end;
```

```
Folders.Add(SharedMessage as SharedNotification, [Name as String])
```

This method creates a shared folder based on the SharedNotification object that is passed in as the first parameter. Name will set the Folder.Name property of the newly created folder. For more information on Shared Folders, see below.

Getting at the Contents of a Folder

One of the main reasons for wanting to access a GroupWise folder is to get at the messages contained within. To that end, look at a folder as the collection of message and subfolders that it contains. To access the messages in a folder, you have several options:

First, you can access the `Folder.Messages` property, which returns all of the messages contained within the folder in a `Messages` collection.

Second, you can use the `Folder.FindMessages()` method.

```
Folder.FindMessages(Condition)
```

This method takes one parameter; a variant called `Condition`. `Condition` may be a string filter expression or a previously created `Filter` object. For more information on `Conditions` and how to create them, see Chapter 8: Understanding Field and Field Related Objects.

Because the `Messages` collection object returned by `Folder.Messages` takes more resources and is slower than a `MessageList` object returned by `Folder.FindMessages()`, you should use `Folder.FindMessages()` when you can. However, if you have already instantiated a `Messages` object using the `Folder.Messages` property, feel free to do so again without concern about performance.

The following sample searches the Mailbox Folder for messages that include the word “the”. It then displays the number of messages found.

Example in VB:

```
'gwAccount is a valid Account object

Dim gwFolders As GroupwareTypeLibrary.Folders
Dim gwFolder As GroupwareTypeLibrary.Folder2
Dim gwMessages As GroupwareTypeLibrary.Messages

Private Sub FindMessagesWithThe()
Set gwFolder = gwAccount.Mailbox
Set gwMessages = gwFolder.FindMessages("(Message contains ""the"")")

Msgbox gwMessages.Count
End Sub
```

Example in Delphi:

```
//gwAccount is a valid Account object

procedure FindMessagesWithThe;
var gwFolder,gwMessages:variant;
begin
gwFolder:=gwAccount.Mailbox;
gwMessages:=gwFolder.FindMessages('(Message contains "the")');
ShowMessage(inttostr(gwMessages.Count));
end;
```

Manipulation of `Messages` and `MessageList` collections are discussed further in Chapter 5: Understanding Message and Message Related Objects and Chapter 6: Understanding Folder and Trash Related Objects.

Deleting a Folder

To delete a folder, use `Folder.Delete()`. All messages in the folder, as well as all subfolders and all messages in those subfolders will be deleted. The messages are moved to the trash folder. Note that if the messages are owned by another account or if the folder is incoming shared, the messages and folder will not be deleted from the owner's account.

The following procedure searches for the first folder named 'MainFolder' and, if it exists, deletes it:

Example in VB:

```
'gwAccount is a valid Account object

Dim gwAllFolders As GroupwareTypeLibrary.Folders
Dim gwMainFolder As GroupwareTypeLibrary.Folder2

Private Sub DeleteFolder()
Set gwAllFolders = gwAccount.AllFolders
Set gwMainFolder = gwAllFolders.ItemByName("MainFolder")
```

```

On Error GoTo FldrErr
    gwMainFolder.Delete
Exit Sub
FldrErr:
    MsgBox "Error Deleting Folder: " & Err.Description & Err.Number
Exit Sub

```

Example in Delphi:

```

//gwAccount is a valid Account object

procedure DeleteFolder;
var gwAllFolders, gwFolder:variant;
MainFolder:string;
begin
    MainFolder:='MainFolder';
    gwAllFolders:=gwAccount.AllFolders;
    try
        gwFolder:=AllFolders.ItemByName(MainFolder);
        gwFolder.Delete;
    except
        ShowMessage('Error Deleting Folder');
    end;
end;

```

Moving a Folder

To move a folder, use `Folder.Move()`. This method takes one parameter called `DestFolder` which must be a valid `Folder` object. The Mailbox, Calendar, and Query folders should not be used. The following code example moves the first folder in the cabinet, if it exists, into the `RootFolder`:

Example in VB:

```

'gwAccount is a valid Account object

Dim gwFolders As GroupwareTypeLibrary.Folders
Dim gwFolder1 As GroupwareTypeLibrary.Folder2
Dim gwFolder2 As GroupwareTypeLibrary.Folder2

Private Sub MoveFolder()
Set gwFolders = gwAccount.Cabinet.Folders
If gwFolders.Count > 0 Then
    Set gwFolder1 = gwAccount.Cabinet.Folders.Item(1)
    Set gwFolder2 = gwAccount.RootFolder
    On Error GoTo FldrErr
    gwFolder1.Move gwFolder2
End If
Exit Sub
FldrErr:
    MsgBox "Error with move: " & Err.Description & Err.Number
End Sub

```

Example in Delphi:

```
//gwAccount is a valid Account object

procedure MoveFolder;
var gwFolder1,gwFolder2:variant;
begin
  if gwAccount.Cabinet.Folders.Count > 0 then
  begin
    gwFolder1:=gwAccount.Cabinet.Folders.Item(1);
    gwFolder2:=gwAccount.RootFolder;
    gwFolder1.Move(gwFolder2);
  end else
    ShowMessage('Error with move:');
end;
```

Renaming a Folder

To change the name of a folder, simply change the `Folder.Name` property. This will not work for incoming shared folders or System folders.

Sharing a Folder

For information on Shared folders, see below.

Shared Folders

GroupWise allows you to share your folders with other users.

Testing a Folder for Shared Status

To test whether a folder is shared, check the `Folder.Shared` property. If the property is `egwNotShared` it is not a shared folder. If the property is `egwSharedOutgoing` the account owner has shared this folder with others. If the property is `egwSharedIncoming` someone else owns the folder and has shared it with the current account owner. The owner of the folder is stored in `Folder.Owner` as an `Address` object.

The following example tests the type of folder of each top-level folder in the Cabinet folder and reports the results:

Example in VB:

```
'gwAccount is a valid Account object

Dim gwFolders As GroupwareTypeLibrary.Folders
Dim gwFolder As GroupwareTypeLibrary.Folder2
Dim gwCurrentFolder As GroupwareTypeLibrary.Folder2
```

```

Private Sub CheckSharedStatus()
'Get the most recent Folders list.
gwAccount.Refresh

Set gwFolders = gwAccount.Cabinet.Folders
If gwFolders.Count > 0 Then
  For i = 1 To gwFolders.Count
    Set gwCurrentFolder = gwFolders.Item(i)
    Status = gwCurrentFolder.Shared
    Select Case Status
      Case "0"
        MsgBox "Folder " & gwCurrentFolder.Name & " is not shared"
      Case "2"
        MsgBox "Folder " & gwCurrentFolder.Name & " is shared by" &
gwCurrentFolder.Owner.EmailAddress
      Case "1"
        MsgBox "Folder " & gwCurrentFolder.Name & " is shared by the
account owner: " & gwCurrentFolder.Owner.DisplayName
    End Select
  Next i
End If
End Sub

```

Example in Delphi:

```

//gwAccount is a valid Account object

procedure CheckSharedStatus;
var i:integer;gwFolders,gwCurrentFolder:variant;
begin
  gwAccount.Refresh; {This is to get the most recent Folders list. See
  "Refreshing" above.}
  gwFolders:=gwAccount.Cabinet.Folders;
  if gwFolders.Count > 0 then begin
    For i:=1 to gwFolders.Count do begin
      gwCurrentFolder := gwFolders.Item(i);
      case gwCurrentFolder.Shared of
        egwNotShared:ShowMessage('Folder '+ gwCurrentFolder.Name+' is not
        shared');
        egwSharedIncoming:ShowMessage('Folder '+ gwCurrentFolder.Name +' is
        shared by ' + gwCurrentFolder.Owner.EmailAddress);
        egwSharedOutgoing:ShowMessage('Folder '+ gwCurrentFolder.Name +' is
        shared by the account owner: '
        + gwCurrentFolder.Owner.DisplayName);
      end;
    end;
  end;
end;

```

Of course, the above code can be modified for nesting either recursively or by using nested for loops.

When working with a Shared folder, the modifications that you can make depend on the folder rights that you have been given.

Testing a Folder For Folder Rights

If your code attempts to perform an operation on a shared folder that the folder's owner has not given to you in the security preferences (such as deleting the folder), the Object API will return an error. To prevent this from happening, you should always test the `Folder.FolderRights` property. `Folder.FolderRights` is an object in and of itself. However, `Folder.FolderRights` is not a `FolderRights` object. Rather it is a `FolderRightsCollection` object. So, `Folder.FolderRights` returns a collection of `FolderRights` objects.

The following are the key properties of `FolderRights`:

Property	Data Type	Description
Address	Address object (R/O)	The Address that represents the user or group that is associated with this specific set of rights
AllowAdd	Boolean (R/W)	TRUE if user may add to folder, FALSE otherwise
AllowDelete	Boolean (R/W)	TRUE if user may delete from folder, FALSE otherwise
AllowModify	Boolean (R/W)	TRUE if user may modify items in folder, FALSE otherwise
BitMask	Integer (R/W)	Mask of the sharing rights for folders. The constants <code>egwAllowDelete</code> (1), <code>egwAllowAdd</code> (2), and <code>egwAllowModify</code> (4) are all combined in the <code>BitMask</code> property, which is also read/write. Thus, to set the sharing rights to <code>AllowModify</code> and <code>AllowDelete</code> , set the <code>BitMask</code> property to <code>egwAllowDelete + egwAllowModify = (5)</code>

Like other collection objects, to access a `FolderRights` object, you must call the `FolderRightsCollection.Item()` method.

```
FolderRightsCollection.Item(Index as Variant)
```

Parameter	Data Type	Description
Index	Variant	Index may be either an integer, a string representing an integer, or an Address object. If Index is an integer, it must be between 1 and <code>FolderRightsCollection.Count</code> . This is only useful if you want to see the rights for each and every user associated with the shared folder. If Index is an Address object, the <code>FolderRights</code> object for that user, if the user is associated with the folder. If the user is not associated with the folder, the Object API will create an OLE exception.

Item() returns the FolderRights object referenced at Index. Note that the use of Address is difficult because the owner may have shared the folder with a different address than you are expecting. For example, a nicknamed person or personal group. If this happens, even Account .Owner, which you would expect to always work because the folder is in the Owner account, will not work. For this reason, using Address does not work very well in GroupWise remote. A work-around if remote operation is necessary is to iterate through all shared users and test for matching display names.

Let's take another look at our test code from the previous section. We can slightly modify this code to check the folder rights of any of the incoming shared folders on the first level of the cabinet (if they exist). The following example tests the different properties of a Shared folder and uses the BitMask property to show how it relates to the different properties of FolderRights:

Example in VB:

```
'gwAccount is a valid Account object

Dim gwFolders As GroupwareTypeLibrary.Folders
Dim gwCurrentFolder As GroupwareTypeLibrary.Folder2
Dim gwFolderRights As GroupwareTypeLibrary.FolderRights

Private Sub CheckFolderRights()
'Get the most recent Folders list.
gwAccount.Refresh

Set gwFolders = gwAccount.Cabinet.Folders
If gwFolders.Count > 0 Then
  For i = 1 To gwFolders.Count
    Set gwCurrentFolder = gwFolders.Item(i)
    Status = gwCurrentFolder.Shared
    On Error GoTo ShareErr
    Select Case Status
      Case "2"
        Set gwFolderRights =
gwCurrentFolder.FolderRights.Item(gwAccount.Owner)
MsgBox gwCurrentFolder.Name & " " & gwFolderRights.Address.DisplayName & " Delete
Allowed: " & CStr(gwFolderRights.AllowDelete) & ", Add Allowed: " &
CStr(gwFolderRights.AllowAdd) & ", Modify Allowed: " &
CStr(gwFolderRights.AllowModify) & ", BitMask: " & CStr(gwFolderRights.BitMask)
    End Select
  Next i
End If
Exit Sub
ShareErr:
  MsgBox "Share Error: " & Err.Description & Err.Number
End Sub
```

Example in Delphi:

```
//gwAccount is a valid Account object

procedure CheckFolderRights;
var i:integer;gwFolders,gwCurrentFolder, gwAddress,gwFolderRights:variant;
begin
  gwAccount.Refresh; {This is to get the most recent Folders list. }
  gwFolders:=gwAccount.Cabinet.Folders;
  if gwFolders.Count > 0 then begin
    For i:=1 to gwFolders.Count do begin
      gwCurrentFolder := gwFolders.Item(i);
      case gwCurrentFolder.Shared of
        egwSharedIncoming:
          begin
            {trap for errors}      try
            {Get the folder rights}  gwFolderRights:=
            gwCurrentFolder.FolderRights.Item(gwAccount.Owner);

            {Show the rights info}   ShowMessage(gwCurrentFolder.Name+' '+
                                     gwFolderRights.Address.DisplayName +
                                     ' Delete Allowed: ' +
                                     truefalse(gwFolderRights.AllowDelete) +
                                     ', Add Allowed: ' +
                                     truefalse(gwFolderRights.AllowAdd) +
                                     ', Modify Allowed: ' +
                                     truefalse(gwFolderRights.AllowModify) +
                                     ', BitMask: ' +
                                     inttostr(gwFolderRights.BitMask));
            except end;
          end;
        end;
      end;
    end;
  end;
end;
end;
```

Sharing a Folder

Creating a shared folder is a bit tricky. When a folder is not shared, the `Folder.FolderRights` collection is empty. Thus, sharing a folder requires you to create a `FolderRights` object for each user you wish to grant folder access. Here is where it gets confusing. The object at issue is called `Folder.FolderRights`, which, as discussed above, is actually a `FolderRightsCollection` object.

To share a folder, execute the `FolderRightsCollection.Add()` method on the `FolderRightsCollection` object returned by the `Folder.FolderRights` property.

```
FolderRightsCollection.Add(Address as Variant, Rights as Long)
```

Parameter	Data Type	Description
Address	Address object	Address object of the user or group you want to share with
Rights	Long	Now the FolderRights.BitMask property makes sense - the Rights Bit-Mask is the same combination of egwAllowDelete (1), egwAllowAdd (2), and egwAllowModify (4) constants, depending on what rights are to be granted.

This method takes two parameters; an Address object that represents the user you are going to share the Folder with and a long value that sets the rights for that user. If you attempt the Add () method on a non-personal folder, you will receive an error.

To finalize sharing, you need to call FolderRightsCollection.Commit(). The Commit() method saves all sharing changes.

```
FolderRightsCollection.Commit(Subject as String,BodyText as String,[TypeCommit as Enum])
```

Parameter	Data Type	Description
Subject	String	Subject of the notification mail sent to the newly shared (or modified) users
BodyText	String	Text of the message the affected user will read
TypeCommit	Enum	This parameter may equal egwDeleted (1), egwModified (2), or egwNew (3). If the parameter is present, only Deleted, Modified, or New users will be committed, and only those users will receive a message.

This method sends a message to any affected users. If the final parameter is omitted, the message and body text is sent to all changed users, and all changes to affected users are committed. If you include TypeCommit, only the affected user types (deleted users, modified users, or newly shared users) will receive the message. If you wish to send a different message to each category of users, you need to call Commit() three times with a different value of TypeCommit for each call.

The following sample code creates a new shared folder in the Cabinet and then shares that folder with the first user in the system address book with add and modify rights. Before testing this, you might wish to change the address to some other user so you don't confuse the first person in your address book with shared folders (you cannot share with the Account.Owner, or we would have written it like that).

Example in VB:

```
'gwAccount is a valid Account object

Dim gwAddress As GroupwareTypeLibrary.Address
Dim gwAllFolders As GroupwareTypeLibrary.Folders
Dim gwNewFolder As GroupwareTypeLibrary.Folder2
Dim gwRightsCollection As GroupwareTypeLibrary.RightsCollection

Private Sub ShareAFolder()
    Set gwAllFolders = gwAccount.AllFolders
    Set gwNewFolder = gwAllFolders.Add("Test Shared Folder")
    Set gwAddress = gwAccount.SystemAddressBook.AddressBookEntries.Item(1)
    Set gwRightsCollection = gwNewFolder.FolderRights 'remember that this is a
collection
    gwRightsCollection.Add gwAddress, egwAllowAdd Xor egwAllowModify
    gwRightsCollection.Commit "New Test Shared Folder", "This is a test shared
folder", egwNew
End Sub
```

Example in Delphi:

```
//gwAccount is a valid Account object

procedure ShareAFolder;
var gwAddress,gwAllFolders,gwNewFolder,gwRightsCollection:variant;
begin
    gwAllFolders:=gwAccount.Cabinet.AllFolders;
    gwNewFolder:=gwAllFolders.Add('Test Shared Folder');
    gwAddress:=gwAccount.SystemAddressBook.AddressBookEntries.Item(1);
    gwRightsCollection:=gwNewFolder.FolderRights; {remember that this is a
collection}
    gwRightsCollection.Add(gwAddress,egwAllowAdd or egwAllowModify);
    gwRightsCollection.Commit('New Test Shared Folder','This is a test shared
folder',egwNew);
end;
```

Accepting a Shared Folder

When a user creates a shared folder, or the Object API calls the `FolderRightsCollection.Commit()` method, a notification message is sent to all affected users. The incoming Mail object will have a `ClassName` property of `'GW.MESSAGE.MAIL.NGW.SHARED.FOLDER.NOTIFY'`. Testing for this `ClassName` will allow you to add the new shared folder using the `Folders.Add()` method. This version of `Add()` is slightly different than the version used to add a personal folder to a `Folders` collection taking one required parameter and one optional parameter.

```
Folders.Add(SharedMessage as Mail,[Name as String])
```

Parameter	Data Type	Description
SharedMessage	Mail object	This must be the Mail object that represents the shared folder notification. Any other Message or Mail object will fail.
Name	String	You may use Name to set the name of the new incoming shared folder to something other than the default name.

If the SharedFolder parameter is not a shared folder notification mail, the Object API will generate an error. Further, the Mail object is deleted as soon as this method is called. You should not make any further calls after calling this method. You should also unassign the variable. Finally, you cannot obtain sharing information from the shared folder notification. Instead, you must call the Add () method first, and then access the properties of the newly created shared folder, as described above.

The following example checks a Mail message to see whether it is a notification and, if so, adds the shared folder to the Cabinet. Note, of course, that any Folders collection, and not just Cabinet .Folders, can be the recipient of the new incoming shared folder.

Example in VB:

```
'gwAccount is a valid Account object and gwMail is a valid Message object

Dim gwMail As GroupwareTypeLibrary.Mail2

Private Sub CheckAndAccept(gwMail)
If gwMail.ClassName = "GW.MESSAGE.MAIL.NGW.SHARED.FOLDER.NOTIFY" Then
    gwAccount.Cabinet.AllFolders.Add gwMail, "Test Shared Folder"
End If
End Sub
```

Example in Delphi:

```
//gwAccount is a valid Account object and gwMail is a valid Message object

procedure CheckAndAccept(gwMail:variant);
begin
    If gwMail.ClassName='GW.MESSAGE.MAIL.NGW.SHARED.FOLDER.NOTIFY' then
        gwAccount.Cabinet.AllFolders.Add(gwMail,'Test Shared Folder');
end;
```

Removing User Access to a Shared Folder

To remove folder sharing rights for a given user, delete the `FolderRights` object for that user. Actually obtaining a reference to the `FolderRights` object for a given individual is not necessarily an easy task. You have two ways to find the user's `FolderRights`. If you have the user's `Address` property, you may call the `FolderRightsCollection.Item()` method and pass the `Address` to the `Item()` method. The return value will be the `FolderRights` object for the desired user, if the user exists.

If you do not know if the user is in the collection, or if you don't have the `Address` object, you may need to iterate through the `FolderRightsCollection`. First, obtain the value of the `Folder.FolderRights.Count` property. Then, iterate through each `FolderRights` object in the `FolderRightsCollection` using the objects `Item()` method with an integer `Index` parameter in the range between 1 and `Folder.FolderRights.Count`. For each `FolderRights` object in the collection, test the `FolderRights.Address` property (which is an `Address` object) or one of the values of `Address` (such as `Address.DisplayName`) to see whether this is the `FolderRights` object of a person you wish to delete. If so, then call the `FolderRights.Delete()` method to delete the user. Finally, re-commit the folder rights by calling `FolderRightsCollection.Commit()`.

Here is the declaration for `Delete()`.

```
FolderRights.Delete()
```

That was complicated; the primary confusion is caused by the fact (as discussed above) that `Folder.FolderRights` is actually a `FolderRightsCollection` object.

The sample code below shows the deletion of a user in practice. We merely extend the `ShareAFolder` procedure to both share and delete the user rights.

Example in VB:

```
'gwAccount is a valid Account object

Dim gwAddress As GroupwareTypeLibrary.Address
Dim gwAllFolders As GroupwareTypeLibrary.AllFolders
Dim gwNewFolder As GroupwareTypeLibrary.Folder2
Dim gwRightsCollection As GroupwareTypeLibrary.RightsCollection
Dim gwRights As GroupwareTypeLibrary.Rights

Private Sub ShareAFolder()
    Set gwAllFolders = gwAccount.AllFolders
    Set gwNewFolder = gwAllFolders.Add("Test Shared Folder")
    Set gwAddress = gwAccount.SystemAddressBook.AddressBookEntries.Item(1)
    Set gwRightsCollection = gwNewFolder.FolderRights
```

```

    gwRightsCollection.Add gwAddress, egwAllowAdd Xor egwAllowModify
    gwRightsCollection.Commit "New Test Shared Folder", "This is a test shared
folder", egwNew
    Set gwRights = gwRightsCollection.Item(gwAddress)
    gwRights.Delete
    gwRightsCollection.Commit "No more sharing with you", "Gone before you know
it",
egwDeleted
End Sub

```

Example in Delphi:

```

//gwAccount is a valid Account object

procedure ShareAFolder;
Var gwAddress,gwAllFolders,gwNewFolder,gwRightsCollection,gwRights:variant;
begin
    gwAllFolders:=gwAccount.Cabinet.AllFolders;
    gwNewFolder:=gwAllFolders.Add('Test Shared Folder');
    gwAddress:=gwAccount.SystemAddressBook.AddressBookEntries.Item(1);
    gwRightsCollection:=gwNewFolder.FolderRights; {remember that this is a
collection}
    gwRightsCollection.Add(gwAddress,egwAllowAdd or egwAllowModify);
    gwRightsCollection.Commit('New Test Shared Folder','This is a test shared
    folder',egwNew);
    gwRights:=gwRightsCollection.Item(Address); {same address we created with}
    gwRights.Delete;
    gwRightsCollection.Commit('No more sharing with you','Gone before you know
    it',egwDeleted);
end;

```

The Trash Object

The Trash object is a special type of Folder object which is returned by Account.Trash. The special properties associated with this GroupWise object are:

Property	Data Type	Description
Name	String (R/O)	The name of the trash folder
Parent	Account (R/O)	The Account to which this trash object belongs
TrashEntries	TrashEntries (R/O)	This object holds the particular deleted items

The Trash folder has two methods.

```
Trash.Empty()
```

This method empties the trash folder.

```
Trash.Refresh()
```

This method rereads the trash data from the database. You should call Refresh() before performing any functions on the trash folder to make sure you have the latest data.

The TrashEntries Object

A TrashEntries collection holds TrashEntry objects (described below). This collection has the same Count property, Item() and Find() methods as other collections. Item() takes an integer between 1 and TrashEntries.Count. Find() takes a Condition that returns a collection of matching TrashEntry objects. In addition, TrashEntries has an additional method called ItemEx().

```
TrashEntries.ItemEx(Index as Variant)
```

Unlike Item(), in ItemEx(), the value of Index is a variant. If Index is an integer, then ItemEx() returns the TrashEntry at that position in the collection. Index may also be a valid MessageID in which case the TrashEntry object with the same MessageID is returned. Finally, Index may be a Message object, in which case the TrashEntry corresponding to that Message object (if it exists in the collection) is returned.

The TrashEntry Object

The TrashEntry object is the actual deleted message that shows up in the GroupWise trash can. The key properties of TrashEntry are:

Property	Data Type	Description
Folder	Folder object (R/O)	This is the folder that contained the message before it was deleted
Message	Message object (R/O)	This is the Message object that corresponds to the trash item.

There are two methods associated with TrashEntry; these methods control deleting and undeleting the trash items.

```
TrashEntry.Delete()
```

This method empties an item from the trash.

```
TrashEntry.Undelete()
```

This method returns the Message object associated with the TrashEntry to the Folder object from which it came.

The following sample uses all of the trash objects described above. First, it obtains a reference to the trash object. Next, it finds all items created more than one day ago. Finally, the code obtains a reference to the first trash message. The code also shows how ItemEx() works.

Example in VB:

```
'gwAccount is a valid Account object

Dim gwTrashCan As GroupwareTypeLibrary.Trash
Dim gwAllTrashItems As GroupwareTypeLibrary.TrashEntries2
Dim gwFoundTrashItems As GroupwareTypeLibrary.TrashEntries2
Dim gwTrashMessage As GroupwareTypeLibrary.TrashEntry
Dim gwTestMessage1 As GroupwareTypeLibrary.TrashEntry
Dim gwTestMessage2 As GroupwareTypeLibrary.TrashEntry

Private Sub RunTrashExample()
Set gwTrashCan = gwAccount.Trash
Set gwAllTrashItems = gwTrashCan.TrashEntries
Set gwFoundTrashItems = gwAllTrashItems.Find("(CREATE_DATE <= YESTERDAY)")

If gwFoundTrashItems.Count > 0 Then
    Set gwTrashMessage = gwFoundTrashItems.Item(1)
    Set gwTestMessage1 = gwFoundTrashItems.ItemEx(gwTrashMessage.Message)
    Set gwTestMessage2 =
gwFoundTrashItems.ItemEx(gwTrashMessage.Message.MessageID)

    MsgBox "Are they the same? String: " &
gwTrashMessage.Message.Subject.PlainText & ", Message: " &
gwTestMessage1.Message.Subject.PlainText & ", MessageID: " &
gwTestMessage2.Message.Subject.PlainText

gwTrashMessage.Undelete
End If
End Sub
```

Example in Delphi:

```
//gwAccount is a valid Account object

procedure RunTrashExample;
var gwTrashCan, gwAllTrashItems, gwFoundTrashItems, gwTrashMessage,
gwTestMessage1, gwTestMessage2:variant; gwTrashMessageID:string;
begin
gwTrashCan:=gwAccount.Trash;{note that this line and the next can be done in
one step}
gwAllTrashItems:=gwTrashCan.TrashEntries;
gwFoundTrashItems:=gwAllTrashItems.Find('(CREATE_DATE <= YESTERDAY)');
if gwFoundTrashItems.count > 0 then
begin
gwTrashMessage:=gwFoundTrashItems.Item(1);
{now is the useless testing to show ItemEx()}
end
end
```

```

    gwTestMessage1:=gwFoundTrashItems.ItemEx(gwTrashMessage.Message);{item by
message object}
    gwTestMessage2:=gwFoundTrashItems.ItemEx(gwTrashMessage.Message.MessageID);
{item by MessageID}
    showmessage('Are they the same? String:
'+gwTrashMessage.Message.Subject.Plaintext
    + ', Message: '+gwTestMessage1.Message.Subject.Plaintext+', MessageID: '
    + gwTestMessage2.Message.Subject.Plaintext); {these should all be the
same}
    {end testing}
    gwTrashMessage.Undelete; {restore from the trash can}
end;
end;

```

Summary

You have learned how GroupWise uses Folders to store information. Using the Object API, you can create, access, modify, and delete many types of Folders. In addition, the Object API provides objects and methods for you to use to manipulate access rights; Read, Add, Edit, Delete for a given user. In the next chapter we look at the collections which hold Messages.

For more information on Folders, please check the GroupWise Object API documentation <http://developer.novell.com/ndk/doc.htm>, sample code <http://developer.novell.com/ndk/sample.htm> or visit the Developer Support Forum area at <http://developer-forums.novell.com/category/index.tpt>.

Copyright © 2002 by Novell, Inc. All rights reserved.
No part of this document may be reproduced or transmitted
in any form or by any means, electronic or mechanical,
including photocopying and recording, for any purpose
without the express written permission of Novell.

All product names mentioned are trademarks of
their respective companies or distributors.

Message Collections

Chapter 4

Section 1: GroupWise Object API

Messages are gathered in message collections. These collections allow you to find, sort, and process messages in many different ways. In the GroupWise Object API, there are three different types of objects that hold message collections.

- Messages object
- MessageList object
- AllMessages object

The following topics are discussed in this chapter.

Contents:

- The Messages Collection
- The AllMessages Collection and the AllMessagesIterator
- Message Lists

The Messages Collection

The `Messages` collection object is similar to the `Folders` collection, except that it holds messages. It consists of all the messages in a folder, and thus the only place to find a `Messages` object collection is by accessing the `Folder.Messages` property. This collection can be very useful for manipulating messages in GroupWise, but it also consumes more resources than the `MessageList` object. Therefore, you should only use this collection when you need to work directly with specific folders.

The following are the key properties and methods for iterating through these collections and accessing specific messages.

The Count Property and Item() Method

The `Count` property is an integer that holds the number of messages in the collection. This property is needed when iterating through the messages in the collection.

Please note, however, that the number of messages in a query (a find results) folder is always “0”. Thus, if you were to check on the “Count” property of the “Sent Items” folder, or the “Task List” folder, or any other query folder that you might have, the value of `Count` would be “0”. This has confused many GroupWise developers.

The reason is simple. There is never anything in a query folder until the folder is accessed. Accessing the folder causes a search to be conducted according to the search criterion of the query folder to fill the folder. Every time you click on it, the search is performed again to display results to the user. But there is nothing in the folder originally, and hence the `Count` value is always “0”.

Use the `Item()` method to obtain a specific message in the collection.

`Item(Index)`

`Item()` takes as single parameter called `Index`, which is a variant. You can make `Index` an integer between 1 and `Messages.Count`, and the `Message` object corresponding to that position will be retrieved. This is useful when iterating in a for loop.

`Item()` can also be used for retrieving specific messages by making `Index` a string. The string you pass must be a valid `MessageID` to one of the `Message` objects in the collection. If `Index` is a valid `MessageID`, the `Message` object with that `MessageID` will be returned.

The following Delphi sample will scroll through all of the messages in the Mailbox's Messages collection and show the subject of each.

```
This assumes gwAccount is valid.

procedure ScrollMessages;
var msgs,themessage:variant;i:integer;
begin
Msgs:=gwAccount.Mailbox.Messages;
  for i:=1 to Msgs.Count do begin
    themessage:=Msgs.Item(i);
    showmessage(themessage.subject.plaintext);
  end;
end;
```

Here is an equivalent example in Visual Basic:

```
This assumes gwAccount is valid.

Dim Msgs As Messages
Dim theMessage As Message

Set Msgs=gwAccount.Mailbox.Messages

For i = 1 To Msgs.Count
  Set theMessage=Msgs.Item(i)
  MsgBox(themessage.subject.plaintext);
Next
```

Finding Messages

Use the Find() method of the Messages object to locate messages within the specific collection that match a search query.

```
Find(Condition)
```

This method takes one variant parameter called `Condition`. `Condition` may be a string filter expression or a previously created `Filter` object. (See Chapter 8: Understanding Field and Field Related Objects about using filter expressions and filter objects to execute queries). The `Find()` method returns a `MessageList` object, which is discussed in the next section.

Let's take another look at `FindMessagesWithThe` procedure from the folders discussion in Chapter 3: Understanding Folder and Trash Related Objects. We can use almost the same code with `Messages.Find()`.

```

procedure FindMessagesWithThe2;
var thefolder,foundmessages:variant;
begin
thefolder:=gwAccount.Mailbox;
foundmessages:=thefolder.messages.find('(Message contains "the")');
ShowMessage(inttostr(foundmessages.count));
end;

```

Here is a Visual Basic example, which gets all the messages in a Mailbox folder and then uses a find operation to find just the incoming mail. It uses late binding (using a Variant type)

This assumes objFolder has already been accessed

```

Dim Msgs As Variant
Dim FilterString As String
Dim MyMsgList As Variant

Set Msgs=objFolder.Messages
FilterString = "(BOX_TYPE = INCOMING AND MAIL)"
Set MyMsgList = Msgs.Find(FilterString)

```

Moving Messages in Messages Collections

One of the primary benefits of the Messages collection is the ability to manipulate messages within the GroupWise database.

Moving Messages. To move messages between Messages object collections – in other words, between GroupWise folders – you should use the `Messages.Move()` method. `Move()` takes two parameters.

```
Messages.Move(Index, Destination)
```

Name	Data Type	Description
Index	Variant	Either an ordinal or a MessageID, just as it is defined in the Item() method. In addition, Index may be a Message object, which will be a reference to a single message
Destination	Messages object	This parameter must be another Messages object where you want to move the message.

The following code will move the first message in the Mailbox to the Work-In-Progress folder.

```

procedure MoveAMessage;
begin
If gwAccount.Mailbox.Messages.Count>0 then
  begin
    gwAccount.Mailbox.Messages.Move(1,gwAccount.WorkFolder.Messages);
  end;
end;

```

Note: The MoveAMessage procedure may move a sent item to the Work-In-Progress folder. This is because the Folder.Messages object contains all of the messages in that folder, including sent items. Thus, if you use an ordinal as the index in methods such as Item() or Move(), you will not be able to control the type of message you get. You can do a query, find, or test each message to limit the types of messages you are working with - as shown in a previous example.

We have included a simple index here for exemplary purposes because we do not know how your users' GroupWise mailboxes are organized.

Adding New Messages to the Message Object Collection

You can also use one of the three Messages.Add() methods to create new messages in the OAPI. These Add() methods can take anywhere from 0-3 arguments, depending on the type of message you wish to create. In addition, a new method called "AddExistingMessage" is now available for GroupWise 6.0 SP1 and GW 5.5 EP SP4. This method can add an already existing message to a GroupWise database, and is also discussed below.

The most commonly used Add method creates a message type of your choice, using the following syntax:

```
Messages.Add([Class],[ObjType])
```

Name	Data Type	Description
Class	String	The Class string is the type of message you wish to create, using the C3PO message string types (see also Message.ClassName above). This parameter may be 'GW.MESSAGE.APPOINTMENT', 'GW.MESSAGE.MAIL', 'GW.MESSAGE.NOTE', 'GW.MESSAGE.PHONE', or 'GW.MESSAGE.TASK', or any custom class you may create. Omitting this parameter assumes the Class string is GW.MESSAGE.MAIL
ObjType	Enumerated Integer	You should use the second parameter, ObjType, if you wish to create a personal message rather than a draft message. If you omit the parameter, the OAPI will assume that you set this parameter to egwDraft (4). You can create a personal message by using the egwPersonal (3) value for the second parameter.

Note that the parameters used by this version of Add are optional. (Don't forget, if you want to get to the second parameter, you must include the first parameter, even if it is blank). With the defaults above, notice that using Add () without any parameters will create a draft "Mail" object.

```
Messages.Add()
```

You can create either a draft message or a personal message, depending upon the value of `ObjType`.

Below is a Visual Basic example that accesses the `WorkFolder`, gets the `Messages` collection in this folder, and then adds a new draft message (of an appointment type) to this collection.

```
This assumes objAcc has already been accessed and all variables have been properly dimensioned.
```

```
Set MyWorkFolder = objAcc.WorkFolder
Set MyMessages = MyWorkFolder.Messages
Set MyDraftMsg = MyMessages.Add("GW.MESSAGE.APPOINTMENT")
MyDraftMsg.Subject = "This is the subject"
MyDraftMsg.BodyText = "This is the body text"
```

Creating and Sending Messages. As shown above, the `Messages.Add()` method can return a new message in the form of a draft message. A typical use for this new draft message is to give the application developer the chance to fill out the various fields of the message (using various message properties such as the `Subject` and `BodyText` above), and then to send it (using `Message.Send()`).

Once the draft message object is sent, it can then be deleted (or should be deleted, depending on your application). If you attempt to use the same draft message again by changing only the `TO` field of the message and sending it out again, you should get an error message.

If you would like the draft message to be automatically deleted, you should create all your new messages in the `Account.WorkFolder.Messages` collection, which corresponds to the `Work-In-Progress` folder in `GroupWise`. Otherwise, you can call the `Message.Delete()` method to delete the draft message after you have called `Message.Send()`, though the message will automatically be linked to the `WorkInProgress` folder as well.

For reasons already discussed, note that the `Add()` method will not work and will return an error if you attempt to call it in a `Messages` collection owned by a query folder object. In other words, you can't add messages to a query folder.

Adding Document References. If you choose to pass the `Add()` method '`GW.MESSAGE.DOCUMENTREFERENCE`', then you will trigger a second version of `Add()` that takes three parameters.

```
Message Add(Class,Referent,[Version])
```

Parameter	Data Type	Description
ClassName	String	For this version of Add, you must pass 'GW.MESSAGE.DOCUMENTREFERENCE' as ClassName
Referent	Object	If you are creating a document reference, the second parameter, Referent, is required. Referent must be either a Document or DocumentVersion object that refers to the document for which you want to create a document reference.
Version	Enum	The third parameter, Version, is an enumerated type that signifies the document version you want to reference. The possible values egwOfficial (0) for official versions, egwCurrent (1) for the current version, and egwSpecific (2) for a specific version.

Just as you might expect, you will want to trigger this version if you want to create a new document reference within the folder that holds this Messages collection. If Referent is a Document object, you should choose either egwOfficial or egwCurrent for Version (if you omit Version altogether, the OAPI will assume a value of egwCurrent). If you want a specific version, on the other hand, Referent should be a DocumentVersion object, and Version should be egwSpecific – a document reference to the specific version will be created.

This method returns the DocumentReference message subclass object.

The following procedure shows how to add a document as a document reference. Ignore for now the code creating the document. We will discuss that later.

```

procedure AddDocument;
var document,docref:variant;
begin
Document:=gwAccount.DefaultDocumentLibrary.Documents.AddEx('c:\config.sys',
    {leave second parameter blank for default},gwAccount.Mailbox);
ShowMessage('New Document Number: '+inttostr(Document.DocumentNumber));
docref:=gwAccount.MailBox.Messages.Add('GW.MESSAGE.DOCUMENTREFERENCE',
    Document, egwCurrent);
ShowMessage(docref.subject.plaintext);
end;

```

Linking Messages. Since it is possible for a message to exist in multiple folders, you can link an already existing message to a new folder. You can do this by using one of the syntax types for the Messages.Add() method.

```

Messages.Add(Message)

```

When linking messages, you pass the method a single parameter called `Message`, which is a `Message` object. Thus to link a message to another folder, you should obtain a reference to the message via the `Item()` method. Once you have this reference, you can pass it to the `Folder.Messages.Add()` method for the new folder.

If, however, the message does not belong to the owner of the current `Account` – for example, if the message was an attachment to an incoming mail message – you will receive an error.

The following procedure will link `Mail`, which is a valid `Message` object, to the `Work-In-Progress` folder.

```
procedure LinkMessage(Mail:variant);
begin
  GWAccount.WorkFolder.Messages.Add(Mail);
end;
```

You should know about one other wrinkle. If you add/link to a shared folder, you may actually wind up creating a copy of the message. This is one of the few times in `GroupWise` when a linked message is actually a copy, and it is caused by the way `GroupWise` stores messages in different databases. To allow control in such cases, the `Add()` method returns a `Message` object. In most cases, the returned object will be the same as the object that you pass in to the method. However, if a copy of the message is created, the method will return the new `Message` object. You should compare the input and output `Message.MessageID` property to determine whether a new message was created.

In addition, exercise caution if you ever need to merge archive accounts (using the `MergeArchive` API) with messages that are linked to other folders. Duplicate messages may be created.

AddExistingMessage. A relative newcomer on the scene is the “`AddExistingMessage`” method made available for `GroupWise 6.0 SP1` and `GW 5.5 EP SP4` (and later versions of each). This new method immediately adds a message to the database, but will not send the message. It can, for example, take an existing `Palm` message and add it to the database like it was received by `GroupWise`. The `GroupWise` system treats these like `InBox` or `OutBox` items, but will not try to send them again.

The syntax for this new method is more complicated than most, and is shown below:

```
Messages.AddExistingMessage(SenderDisplayName, SenderEmailAddress,
SenderEmailAddressType, CreationDate, MessageBoxType, MessageStatus,
MessagePriority, MessageSecurity, [DraftMessage], [LastModificationData])
```

Parameter	Data Type	Description
SenderDisplayName	String	The display name of the person who sent the message
SenderEmailAddress	String	The email address of the person who sent the message
SenderEmailAddressType	String	The email address type of the person who sent the message
CreationDate	DATE	The date you wish to report as the creation date of the message
MessageBoxType	Enumerated Integer	The type of message. See the MessageBoxTypeConstants
MessageStatus	Enumerated Integer	A bitmask that defines the status of the message. See the MessageStatusConstants
MessagePriority	Enumerated Integer	The priority of the message. See the MessagePriorityConstants
MessageSecurity	Enumerated Integer	The security of the message. See the MessageSecurityConstants
DraftMessage	Object	Optional - the message object you want to insert into your database, with its properties filled out, such as body text.
LastModificationdate	Object	Optional - date and time of last modification

Here is a complete example of using the AddExistingMessage method in Visual Basic. This example uses late binding. There is also some commented out code represents an “incorrect” way to use the AddExistingMessage method. You may have partial success with the commented out code, but it will not allow you to specify any body text, recipients, etc. for the message. The last few lines simply show the new message, borrowing some code from the GroupWise token API.

```

Dim GW As Variant
Dim Acc As Variant
Dim MyMailFolder As Variant
Dim MyMessages As Variant
Dim MyMessage As Variant
Dim MyDraftMsg As Variant
Dim MyRecips As Variant
Dim MyRecip As Variant

Dim SenderDisplayName As String
Dim SenderEmailAddress As String
Dim CreationDate As Date
Dim iCount As Integer

Dim vCommander As Variant
Dim ParamStr As String
Dim MyMsgID As String
Dim sResult As String
Dim iRet As Integer

```

```

Set GW = CreateObject("NovellGroupWareSession")
Set Acc = GW.Login

Set MyMailFolder = Acc.MailBox
Set MyMessages = MyMailFolder.Messages

` How many messages are there?
  iCount = MyMessages.Count

` Incorrect method. Doesn't give any subject, recipients, etc.

` SenderDisplayName = "John Doe"
` SenderEmailAddress = "jdoe.GWPost.GWDomain"
` SenderEmailAddressType = "NGW"
` CreationDate = "12/25/2000 8:00:00 AM"

` Set MyMessage = MyMessages.AddExistingMessage(SenderDisplayName, ` `
` SenderEmailAddress, SenderEmailAddressType, CreationDate, egwIncoming,
` egwMessageDelivered, egwNormal, egwDefaultSecurity)

` Correct method. Use a draft message.

Set MyDraftMsg = MyMessages.Add("GW.MESSAGE.MAIL")
MyDraftMsg.Subject = "My Subject"
MyDraftMsg.BodyText = "Adding a new message without sending it"

MyDraftMsg.FromText = Acc.RootFolder.Name

Set MyRecips = MyDraftMsg.Recipients
Set MyRecip = MyRecips.AddByDisplayName("Mary Doe")
Set MyRecip = MyRecips.AddByDisplayName("Fu Ling Yu")
Set MyRecip = MyRecips.AddByDisplayName("Santa Claus")

SenderDisplayName = "John Doe"
SenderEmailAddress = "jdoe.GWPost.GWDomain"
SenderEmailAddressType = ""
CreationDate = "12/25/2000 8:00:00 AM"

Set MyMessage = MyMessages.AddExistingMessage(SenderDisplayName,
  SenderEmailAddress, SenderEmailAddressType, CreationDate, egwIncoming,
  egwMessagePrivate, egwNormal, egwDefaultSecurity, MyDraftMsg)

MyMailFolder.Refresh
Set MyMessages = Nothing
Set MyMessages = MyMailFolder.Messages

` This value should be one more than it was before
  iCount = MyMessages.Count

` Open the new message to view it
  MyMsgID = MyMessage.MessageID
  Set vCommander = CreateObject("GroupWiseCommander")
  ParamStr = "ItemOpen("" + MyMsgID + "")"
  iRet = vCommander.Execute(ParamStr, sResult)

```

Remove Method. The one remaining method of the Messages Object that we have not discussed yet is the Messages.Remove() method. As the name suggests, this method removes a message from the message collection where it is located.

This method takes one variant argument that allows it to remove a message in various ways. First, it can take an index which can vary from 1 to Messages.Count. This removes the message at that location in the order the messages are stored in the collection. A second type of argument can be a MessageID string identifying the message to be deleted. Finally, the third type of argument can be the message object itself that we want to delete.

Note: If you attempt to delete the first item in a Messages collection (or any type of GroupWise collection for that matter), all the other messages are shifted down by 1 so that old Item(2) is now Item(1). If you attempt to delete multiple messages by using the "original" item values, you will end up not deleting the items you thought you were deleting.

The AllMessages Collection and the AllMessagesIterator

The AllMessages collection is a special collection that holds all messages in an account, in all folders, whether hidden or not. This collection is accessible only as the Account.AllMessages property. The collection does not include messages owned by another user's account (note the difference between this and AllFolders, which does include shared folders owned by other users' accounts). Thus, if you wish to access shared messages (including document references), you will need to use a different means, such as shared folder access or performing a query.

Notably, because the AllMessages collection can be a very large collection, it does not include a count property. Instead, to access the messages in an AllMessages collection, you should instantiate an AllMessagesIterator object by calling the highly creative AllMessages method called CreateAllMessagesIterator(). As noted, this method takes no parameters

```
AllMessagesIterator := AllMessages.CreateAllMessagesIterator()
```

Using the AllMessagesIterator you can scroll through all of the messages in the Account. The methods for AllMessagesIterator are listed in the following table.

Name	Description
Next()	This takes no parameters, and returns the next Message object in the collection.
Reset()	This takes no parameters, and returns the iterator to before the first message, so that calling Next() will return the first message.

Name	Description
Skip(NumMessages)	This takes one long integer parameter called NumMessages. Skip(NumMessages) will cause the iterator to skip over NumMessages messages. This method is mostly useless, for two reasons. First, it returns an error if the number to skip places the iterator beyond the last message, yet it is impossible to test this because there is no count property. Second, because the messages are in no particular order, you will have no idea what you are skipping.
Clone()	This method will create a copy of the AllMessagesIterator with the iterator at the same position as the cloned object. This makes it possible to keep track of multiple iteration positions, and go back if necessary.

The following Delphi procedure shows how the AllMessage iterators works in practice:

```

procedure IterateMessages;
var Messages, Iterator, Iterator2, TheMessage: variant; docnum: integer;
begin
Messages := gwAccount.AllMessages;
Iterator := Messages.CreateAllMessagesIterator;
TheMessage := Iterator.Next;
ShowMessage('Subject: '+TheMessage.Subject.PlainText);
Iterator.Skip(2);
TheMessage := Iterator.Next;
ShowMessage('Skipped two, so now on fourth message: ' +
TheMessage.Subject.PlainText);
Iterator2 := Iterator.Clone;
Iterator.Reset;
TheMessage := Iterator.Next;
ShowMessage('Iterator1, back to the beginning: '+TheMessage.Subject.PlainText);
TheMessage := Iterator2.Next;
ShowMessage('Iterator2, next message: '+TheMessage.Subject.PlainText);
end;

```

Here is an example in Visual Basic (assumes GWAccount has already been dimensioned and accessed). It simply “does something” if the message BoxType is 1, meaning it is an incoming message.

```

Dim AllMessages As Variant
Dim MyAllMsgsIterator As Variant
Dim Msg As Variant

Set AllMessages = gwAccount.AllMessages
Set MyAllMsgsIterator = AllMessages.CreateAllMessagesIterator
Set Msg = MyAllMsgsIterator.Next

While Not Msg Is Nothing
If (Msg.BoxType = 1) Then
` do something
End If
Set Msg = MyAllMsgsIterator.Next
Wend

```

You can also access messages in the `AllMessages` collection by using its `Find()` method.

`Find(Condition)`

This method takes one variant parameter called `Condition`. `Condition` may be a string filter expression or a previously created `Filter` object. The `Find` method works the same way as the `Find` method for the `Messages` collection, which was described earlier in this chapter. See also Chapter 8: Understanding Field and Field Related Objects for additional discussion about using filters and filter expressions. The `Find()` method returns a `MessageList` object, and is more efficient than instantiating a `Messages` collection for one or more folders to perform a search.

The following procedure will find all incoming messages in the user database, and show the number of incoming messages found.

```
procedure FindIncomingMessages;
var allmessages,foundmessages:variant;
begin
allmessages:=gwAccount.AllMessages;
foundmessages:=allmessages.find('(BOX_TYPE=INCOMING)');{result is a messagelist}
ShowMessage(inttostr(foundmessages.count));
end;
```

Message Lists

The `MessageList` object is similar to the `Messages` collection, except that the `MessageList` is independent of the GroupWise message database. Think of this object as a “scratch pad” that keeps track of messages without affecting GroupWise. Other than its disjunction from GroupWise message store, `MessageList` behaves nearly identically as the `Messages` collection. Thus, the key properties and methods follow.

The Count Property and Item() Method

This `Count` property is an integer that holds the number of messages in the collection. This property is needed when iterating through the messages in the collection. Use the `Item()` method to obtain a specific message in the collection.

`MessageList.Item(Index)`

`Item()` takes as single parameter called `Index`, which is a variant. You can make `Index` an integer between 1 and `MessageList.Count`, and the `Message` object corresponding to that position will be retrieved. This is useful when iterating in a for loop.

`Item()` can also be used for retrieving specific messages by making `Index` a string. The string you pass must be a valid `MessageID` to one of the `Message` objects in the collection. If `Index` is a valid `MessageID`, the `Message` object with that `MessageID` will be returned.

Finding Messages

Use the `Find()` method to locate messages within the specific collection that match a search query.

```
MessageList.Find(Condition)
```

This method takes one variant parameter called `Condition`. `Condition` may be a string filter expression or a previously created `Filter` object. The `Find` method works the same way as the `Find` method for the `Messages` and `AllMessages` collections, which was described earlier in this chapter. See Chapter 8: Understanding Field and Field Related Objects for additional discussion about using filters and filter expressions. The `Find()` method returns another `MessageList` object with the matching messages.

Adding Messages to the MessageList

Use the `Add()` method to add messages to the `MessageList`.

```
MessageList.Add(AddedMessage)
```

This method differs from `Messages.Add` because with `MessageList.Add()`, the added message(s) must exist already. `MessageList.Add()` does not create a new message.

This method takes a single variant parameter called `AddedMessage`. If `AddedMessage` is a `Message` object, then that message will be added to the `MessageList`. If `AddedMessage` is another `MessageList`, then all the messages in that list will be added to this list. This `Add` method can also take a `Message ID` as a string. The message with the corresponding ID will be added to the `MessageList`.

The following code combines the `FindMessagesWithThe` procedure and the `FindIncomingMessages` procedure into a single message list using the `MessageList.Add()` method. It then finds all outgoing messages in the new combined message list using the `MessageList.Find()` method.

```
procedure combinedfind;
var foundmessages,foundmessages2,foundmessages3:variant;
begin
foundmessages:=gwAccount.AllMessages.Find('(Message contains "the")');
ShowMessage('Messages with the: '+inttostr(foundmessages.count));
foundmessages2:=gwAccount.AllMessages.Find('(BOX_TYPE=INCOMING)');
ShowMessage('Incoming messages: '+inttostr(foundmessages2.count));
{found messages and foundmessages2 are MessageList objects}
foundmessages.add(foundmessages2); {add one list to the other}
ShowMessage('Combined messages: '+inttostr(foundmessages.count)+' Note that this
    may be smaller than the first two counts due to overlap');
foundmessages3:=foundmessages.find('(Message contains "the")');
ShowMessage('Outgoing messages: '+inttostr(foundmessages3.count) +' This is the
amount of overlap');
end;
```

Removing Messages from the MessageList

Use `Remove ()` to remove a message from the list.

`Remove (Index)`

`Remove ()` takes as single parameter called `Index`, which is a variant. You can make `Index` an integer between 1 and `MessageList.Count`, and the `Message` object corresponding to that position will be retrieved. You may also make `Index` a string. If `Index` is a valid `MessageID`, the `Message` object with that `MessageID` will be removed from the list. Finally, you can make `Index` a specific `Message` object, and that message will be removed from the list.

Summary

In this chapter you learned that messages are gathered in message collections. Using message collections allow you to find, sort, and process messages in many different ways. In the next chapter we will discuss how your application can create, delete, and access the information stored in various properties of a GroupWise message using the object API.

For more information on Messages, please check the GroupWise Object API documentation <http://developer.novell.com/ndk/doc.htm>, sample code <http://developer.novell.com/ndk/sample.htm> or visit the Developer Support Forum area at <http://developer-forums.novell.com/category/index.tpt>.

Copyright © 2002 by Novell, Inc. All rights reserved.
No part of this document may be reproduced or transmitted
in any form or by any means, electronic or mechanical,
including photocopying and recording, for any purpose
without the express written permission of Novell.

All product names mentioned are trademarks of
their respective companies or distributors.

Understanding Message and Message Related Objects

Chapter 5

Section 1: GroupWise Object API

The Message class in GroupWise is the base type for all other messages e.g. Appointment, DocumentReference, Mail, Note, PhoneMessage, SharedNotification, and Task. As a result, the properties and methods found in Message can also be found in all message subtypes. In addition, each subtype has a few unique features that can only be used when you have a subtype of that specific class.

An Attachment object, part of an Attachments collection, is used to identify the files or messages that should be sent as part of an outgoing message. Likewise, a Recipient object, part of a Recipients collection, is used by a GroupWise message to identify the receiver.

The following topics are discussed in this chapter.

Contents:

- Messages
- The Different Types of Message
- Attachments
- Recipients

Messages

Put simply, GroupWise would not exist without the Message object, nor would anyone want to use it without messages. Messages are the base object class for most information transfer in GroupWise. The following message subtypes are derived from the Message object: Mail, Appointment, DocumentReference, Note, PhoneMessage, and Task.

For information on the different object collections, properties, and methods that return a Message object, see Chapter 4: *Message Collections*.

Using the Object API, it is easy to access many useful properties which correspond to the Mail, Appointment, Task, and other Message subtypes that are familiar to most GroupWise users.

The Message Object

The key properties of the Message object are:

Property	Data Type	Description
Attachments	Attachments object (R/O)	The collection of Attachment objects that belong to the message. (See Attachments later in the chapter)
BodyText	Formatted Text object (R/O)	This string property holds a maximum of 32,000 characters in the message body text. It is a FormattedText object and can be accessed with either BodyText.PlainText or BodyText.RTF. The default is PlainText and, as you might have predicted, returns only the text while RTF returns the message body in rich text (.rtf) format.
BoxType	Enum (R/O)	The box type of the message. [egwIncoming (1), egwOutgoing (2), egwPersonal (3), egwDraft (4)]
ClassName	String (R/O)	Identifies the type of message that is represented by the Message object you are working with. Common options are: 'GW.MESSAGE.APPOINTMENT', 'GW.MESSAGE.DOCUMENTREFERENCE', 'GW.MESSAGE.MAIL','GW.MESSAGE.NOTE', 'GW.MESSAGE.PHONE', 'GW.MESSAGE.TASK'. In addition, using a Custom 3rd-Party Object (C3PO) it is possible to create your own custom contexts e.g. 'GW.MESSAGE.MAIL.PROJECT.ABC'
CreationDate	Date (R/O)	Date and time that the message was created.
Deleted	Boolean (R/O)	TRUE if the message has been deleted.
DownloadStatus		

Property	Data Type	Description
EnclosingFolders	Folders object (R/O)	The collection of Folder objects that contain the message.
ExpandedRecipients	Recipients object (R/O)	An expanded RecipientsCollection object that contains all addresses and all members of a group to which the message is addressed.
Fields	Fields object (R/O)	The collection of custom Field objects that belong to the message.
FromText	String (R/O)	Display name that appears in the From field of the message.
MessageID	String (R/O)	Represents the unique identifier of the underlying incoming or outgoing message.
ModifiedDate	Date (R/O)	Date and time that the message was last modified.
NotifyWhenDeleted	Enum (R/W)	Specifies the type of notification that should be sent when the message is deleted. [egwNoNotify (0), egwSendReceipt (1), egwNotify (2), egwSendAndNotify (3)]
NotifyWhenOpened	Enum (R/W)	Specifies the type of notification that should be sent when the message is opened. [egwNoNotify (0), egwSendReceipt (1), egwNotify (2), egwSendAndNotify (3)]
Opened	Boolean (R/O)	TRUE if the message has been opened. You will create an error if you attempt to set this property to FALSE.
Priority	Enum (R/W)	Specifies the priority that should be given to the message. [egwLow (1), egwNormal (2), egwHigh (3)]
Private	Boolean (R/W)	TRUE if the message has been marked private.
Read	Boolean (R/O)	TRUE if the message has been read. You may set this property to FALSE in order to make the message appear unread.
Recipients	Recipients object (R/O)	The RecipientsCollection object that contains the addresses to which the message will be sent. Use one of the Add() methods of the recipients collection to add additional users and groups to the message To, BC, or CC list.
ReplyChildren	MessageList object (R/O)	The collection of Message objects that are next in the reply thread.
ReplyDaysRequested	Long (R/W)	Number of days within which a reply has been requested.
ReplyParent	Message object (R/O)	The Message object that appears previous to the current object in the reply thread.

Property	Data Type	Description
ReplyRequested	Enum (R/W)	Specifies if and when a reply has been requested. [egwNoReply (0), egwWhenConvenient (1), egwWithinDaysRequested (2)]
ReplyRoot	Message object (R/O)	Corresponds to the first message in a reply thread.
Routed	Boolean (R/W)	TRUE if the message has been routed.
RoutingEndOfLine	Boolean (R/O)	TRUE if the message has reached the last user in the list of recipients.
Sender	Address object (R/O)	Corresponds to the message sender. In most cases, Sender.DisplayName will be the same as FromText.
Subject	Formatted Text object (R/O)	Holds the subject as a FormattedText object.

A few additional properties for Message may be found in the official GroupWise Object API documentation.

Sending, Forwarding, or Replying to a Message

Note: Because you would never actually use 'Message' in your applications, replace "MessageSubtype" below with the actual message subtype that you intend to use.

Use the Send() method on a Message object subtype to send a Mail, Appointment, or Task.

```
MessageSubtype.Send()
```

This method requires no parameters and returns the OutBox message after the message is sent. Send() will delete the draft message. If a recipient object fails to resolve, this method will throw an exception. Finally, this method will not update the Frequent Contacts address book. This should be done manually by the caller of the method.

Use the Forward() method on a Message object subtype to forward a Mail, Appointment, or Task.

```
MessageSubtype.Forward()
```

This method takes no parameters and returns a new draft Mail object with the Message object that called it included as an attachment. The new message is contained in the same folders as the original.

Use the Reply() method on a Message object subtype to reply to a Mail, Appointment, or Task.

```
MessageSubtype.Reply([ClassName as String], [ReplyToAll as Boolean],  
[IncludeSenderMessageText as Boolean], [AttachOriginalMessage as Boolean])
```

This method takes up to four parameters and creates a reply message by returning a new draft Message object item with the message sender as the recipient. ClassName, a string, specifies the type of message that GroupWise will create. If ClassName is omitted, "GW.MESSAGE.MAIL" is assumed. ReplyToAll, a Boolean, will include all recipients of the message, except the current user, to the recipients collection of the new draft message if set to TRUE. IncludeSenderMessageText, a Boolean, will include the BodyText of the original message in the BodyText property of the reply message if set to TRUE. AttachOriginalMessage, a Boolean, will add the original message to the Attachments collection of the reply message if set to TRUE.

Any omitted Boolean parameters default to FALSE. The reply message is contained in the same folders as the original and is not automatically linked to the Work In Progress folder (unless the original message was linked there).

Copying a Message

Use the Clone() method on a Message object subtype to duplicate a Mail, Appointment, or Task.

```
MessageSubtype.Clone()
```

This method requires no parameters. Clone() makes an independent copy of the message and returns it as a new draft message. The new message is contained in the same folders as the original.

Refreshing a Message

Use the Refresh() method on a Message object subtype to recursively refresh the Attachments, Fields, Recipients collection on a Mail, Appointment, or Task. In addition, the Message's BodyText, Subject, Address, ReplyChildren/Parent/Root properties are also refreshed.

```
MessageSubtype.Refresh()
```

This method requires no parameters. Refresh() forces the message, associated objects and collections to reread property values from the message database. The actual reading of a specific property may be postponed until the next time the property is accessed. This "lazy evaluation" is an optimization that avoids unnecessary reading of unaccessed properties. If the message is an attachment, it is refreshed when its associated Attachment object is refreshed.

Deleting or Retracting a Message

Use the Delete() method on a Message object subtype to delete a Mail, Appointment, or Task.

```
MessageSubtype.Delete()
```

This method requires no parameters. Delete() moves a message from its associated folder(s) to the trash. This can be somewhat confusing when deleting messages from a shared folder as the message is placed in the owner's trash and not necessarily into the trash of the account that made the delete call.

Use the Retract() method on a Message object subtype to retract a Mail, Appointment, or Task.

```
MessageSubtype.Retract()
```

This method requires no parameters. Retract() will delete a message from a recipient's mailbox provided that the message's BoxType is egwOutgoing, the message has not been opened and has not gone through a gateway to another mail system.

```
MessageSubtype.Annotate(Note as Note)
```

This method adds an existing personal Note object as an attachment to this message. Works even if this message has been sent, because the Note is personal. See also the Add method in the Attachment object. Annotating an item requires create and modify rights to the folder that contains the item. Annotating encapsulated items is not allowed because no distinct containing folders exist for such an item. Annotations can be edited only by the user who originally created the annotation (subject also to folder rights). For example, an annotation created on an item in a shared folder can only be edited by the user who created the annotation.

The Different Types of Message

GroupWise supports six main subtypes of the Message object. In fact, because the Message object is never itself created, one of the objects described below will always be used where a message object is required. Message subtypes include: Mail (standard “e-mail” messages), Appointments, DocumentReferences, Notes, Phone Messages, and Tasks. In addition to the properties and methods that each object inherits from the GroupWise Message object, a Message subtype may also have its own special properties and possibly methods. For received messages, you can read these properties, and you can read or write them for draft messages.

Mail

The Mail object represents a GroupWise Mail “e-mail” message. Mail includes only a couple of extra properties in addition to those that it gets from Message:

Property	Data Type	Description
Completed	Boolean (R/W)	TRUE if the message is completed and can be sent to next individual in a routing slip
Delegated	Boolean (R/O)	TRUE if the message was delegated.
NotifyWhenCompleted	Enum (R/W)	Specifies the type of notification to send when the message is marked completed. [egwNoNotify (0), egwSendReceipt (1), egwNotify (2), egwSendAndNotify (3)]

The Mail object includes one method, in addition to those that it inherits from Message:

```
Mail.Delegate()
```

This method requires no parameters. Delegate() delegates the message to another user and returns the OutBox mail message.

Appointment

The Appointment object represents a GroupWise Meeting or Posted (Personal) Appointment. The simplified hierarchy of an Appointment in GroupWise is:

```
Account object
  Folders collection
    Messages collection
      Appointment object
        BusySearchResults object
          BusySearchElements collection
            BusySearchElement object
              TimeBlocks collection
                TimeBlock collection
```

The properties of an Appointment are:

Property	Data Type	Description
Accepted	Boolean (R/O)	TRUE if the appointment is accepted.
AlarmProgram	String (R/W)	Identifies the Program to execute when the alarm time is reached.
AlarmTime	Date (R/W)	Time when AlarmProgram will execute.
AutoDate	Boolean (R/O)	TRUE if the appointment has an auto-date.

Property	Data Type	Description
AutoDateMessages	MessageList object (R/O)	The MessageList that contains each Appointment object in a string of auto-dated messages.
BusySearchResult	BusySearchResult (R/O)	The busy search results for a draft appointment. Returns nothing if StartBusySearch has never been called or if this is a draft appointment.
BusyType	Enum (R/W)	The type of block this time element represents. [egwFree (0), egwBlocked (1), egwOutOfOffice (2), egwTentative (3)]
Delegated	Boolean (R/O)	TRUE if this appointment has been delegated.
Duration	Double (R/W)	This is the duration in days between EndDate and StartDate. The fractional portion represents the fraction of a day. Duration is considered fixed. If either StartDate or EndDate changes, the other will be changed to keep Duration constant. If Duration changes, then EndDate will be changed to match.
EndDate	Date (R/W)	Marks the end of the appointment.
NotifyWhenAccepted	Enum (R/W)	Specifies the type of notification that should be sent when the appointment is accepted. [egwNoNotify (0), egwSendReceipt (1), egwNotify (2), egwSendAndNotify (3)]
NotifyWhenDeclined	Enum (R/W)	Specifies the type of notification that should be sent when the appointment is declined. [egwNoNotify (0), egwSendReceipt (1), egwNotify (2), egwSendAndNotify (3)]
OnCalendar	Boolean (R/W)	TRUE if the appointment should appear on the Calendar
Personal	Boolean (R/O)	TRUE if the appointment is a personal appointment.
Place	String (R/W)	The location of the appointment.
StartDate	Date (R/W)	Marks the beginning of the appointment.

The Appointment object contains several methods in addition to those that it inherits from Message.

Accepting, Delegating, or Declining an Appointment. Use the Accept() method to accept an Appointment.

```
Appointment.Accept([Comment as String],[AllInstances as Boolean])
```

Although this method requires no parameters, Accept will use the first parameter, Comment, if it exists as the text you want to send in a reply that accepts the appointment. The second parameter, if it exists and is TRUE, will accept all instances of an auto-dated appointment. The default for this value is FALSE or if Appointment is not auto-dated, AllInstances will be ignored altogether.

Use the Delegate() method to delegate an Appointment.

```
Appointment.Delegate([AllInstances as Boolean])
```

This method returns a draft appointment upon which you can set the Recipients and call Send(). Set AllInstances to TRUE if you wish to delegate all instances of an auto-dated appointment and FALSE if you wish to delegate only one instance.

Use the Decline() method to decline an Appointment.

```
Appointment.Decline([Comment as String],[AllInstances as Boolean])
```

As in the previous method, Decline() also requires no parameters. Decline() will however use Comment, if it exists, as the text you want to send in a reply that declines the appointment. Likewise, the second parameter, if it exists and is TRUE, will decline all instances of an auto-dated appointment. Again, if Appointment is not auto-dated, AllInstances will be ignored. The default for this value is also FALSE.

Performing Busy Searches for an Appointment. Use the StartBusySearch() method to check the availability of Recipient objects that you have added to the Appointment.Recipients collection.

```
Appointment.StartBusySearch([StartDate as Date], [Range as Long])
```

This method uses the specified StartDate, or Appointment.StartDate if StartDate is omitted, to return a BusySearchResult object that meets the specified Range (7 or greater). Because BusySearchResult may return with a partial snapshot of elements before it completes, check BusySearchResult.Completed within a while loop using BusySearchResult.Refresh() to make sure you have the full collection of BusySearchElements.

Use the standard Count and Item() practice to iterate through and access the individual items of the BusySearchElements collection that is returned by BusySearchResult.CombinedResult. Using BusySearchElement.FreeBlocks, will return the collection of free time blocks associated with the search result in the form of a TimeBlocks collection. As before, use Count and Item() to access the individual StartDate and EndDate properties of the TimeBlock objects in the collection.

DocumentReference

For information on DocumentReferences, see Chapter 6: *Understanding Message and Message Related Objects*.

Note

The Note object represents a GroupWise Reminder Note. The properties of a Note are:

Property	Data Type	Description
Accepted	Boolean (R/O)	TRUE if the note is accepted.
AutoDate	Boolean (R/O)	TRUE if the note has an auto-date
AutodateMessages	MessageList object	The MessageList that contains each Note object in a string of auto-dated messages.
Completed	Boolean (R/W)	TRUE if the note is completed and can be sent to next individual in a routing slip
Delegated	Boolean (R/O)	TRUE if the note has been delegated from someone else.
NotifyWhenAccepted	Enum (R/W)	Specifies the type of notification to send when the note is accepted. [egwNoNotify (0), egwSendReceipt (1), egwNotify (2), egwSendAndNotify (3)]
NotifyWhenCompleted	Enum (R/W)	Specifies the type of notification to send when the note is marked completed. [egwNoNotify (0), egwSendReceipt (1), egwNotify (2), egwSendAndNotify (3)]
NotifyWhenDeclined	Enum (R/W)	Specifies the type of notification to send when the note is declined. [egwNoNotify (0), egwSendReceipt (1), egwNotify (2), egwSendAndNotify (3)]
OnCalendar	Boolean (R/W)	TRUE if the note should appear on the Calendar
Personal	Boolean (R/O)	TRUE if the note is personal.
StartDate	Date (R/W)	Date when the note will first appear in a recipient's mailbox.

The Note object contains three methods in addition to those that it inherits from Message. These methods: `Accept()`, `Decline()`, and `Delegate()` behave just like the corresponding methods of the Appointment object (see above).

PhoneMessage

The PhoneMessage object represents a GroupWise Phone Message. The properties of a PhoneMessage are:

Property	Data Type	Description
CallerName	String (R/W)	The caller's name
CallerCompany	String (R/W)	The name of the caller's company
CameToSee	Boolean (R/W)	TRUE if the person came to see you
PhoneNumber	String (R/W)	The caller's phone number
PleaseCall	Boolean (R/W)	TRUE if the caller wants you to return their call
ReturnedCall	Boolean (R/W)	TRUE if the caller returned your phone call
Telephoned	Boolean (R/W)	TRUE if the caller telephoned
Urgent	Boolean (R/W)	TRUE if the call is urgent
WantsToSee	Boolean (R/W)	TRUE if the caller wants to see you
WillCall	Boolean (R/W)	TRUE if the caller will contact you again

A PhoneMessage object does not define any methods of its own.

SharedNotification

For information on SharedNotifications, see Chapter 3: *Understanding Folder and Trash Related Objects*.

Task

The Task object represents either a posted or group Task in GroupWise. The properties of a Task are:

Property	Data Type	Description
Accepted	Boolean (R/W)	TRUE if the task is accepted, otherwise FALSE.
AssignedDate	Date (R/W)	Date when the task was assigned.
AutoDate	Boolean (R/W)	TRUE if the task is one of many auto-dated tasks.
AutoDateMessages	MessageList object (R/W)	The MessageList collection that contains each Task object in a string of auto-dated messages.
Completed	Boolean (R/W)	TRUE if the task is completed. Your code may set this to true or false at any time.
Delegated	Boolean (R/W)	TRUE if the task has been delegated from someone else.

Property	Data Type	Description
DueDate	Date (R/W)	Date when the task is due.
NotifyWhenAccepted	Enum (R/W)	Specifies the type of notification that should be sent when the task is accepted. [egwNoNotify (0), egwSendReceipt (1), egwNotify (2), egwSendAndNotify (3)]
NotifyWhenCompleted	Enum (R/W)	Specifies the type of notification that should be sent when the task is completed. [egwNoNotify (0), egwSendReceipt (1), egwNotify (2), egwSendAndNotify (3)]
NotifyWhenDeclined	Enum (R/W)	Specifies the type of notification that should be sent when the task is declined. [egwNoNotify (0), egwSendReceipt (1), egwNotify (2), egwSendAndNotify (3)]
OnCalendar	Boolean (R/W)	TRUE if the task should appear on the Calendar
Personal	Boolean (R/W)	TRUE if the task is personal
StartDate	Date (R/W)	Date when the task should be started. The task will not appear on the recipient's Calendar until the StartDate.
TaskCategory	String	Task category must be a single character with whatever meaning the message sender wants to assign. You will cause an error if you attempt to assign more than one character to the string.
TaskPriority	Integer (R/W)	Task priority can have whatever meaning the message sender wants to assign. You may enter a character, a number, or a character followed by a number.

The Task object contains four methods in addition to those that it inherits from Message. The first three: `Accept()`, `Decline()`, and `Delegate()` behave just like the corresponding methods of the Appointment object (see above).

The fourth method, `MoveToMasterTaskList()`, has the following definition:

```
Task.MoveToMasterTaskList()
```

This method moves the task to the master task list.

Setting and Accessing the Properties of a Message

The following code creates a duplicate of the last message in the Mailbox and sends it to the owner of the account. Note how the method for reading properties is the same as setting them — simply reference the property as a variable. Note also how this sample takes advantage of the different `ClassName` values to check appropriate properties for the message type. This code is also a brief introduction into creating new messages. Note also that `Message.Clone()` would do most of this in one call.

Example in VB:

```
'gwAccount is a valid Account object and the last message is one of the above
message types
```

```
Dim gwMailBox As GroupwareTypeLibrary.Folder
Dim gwSentMsg As GroupwareTypeLibrary.Message
Dim gwOldMsg As GroupwareTypeLibrary.Message
Dim gwNewMsg As GroupwareTypeLibrary.Message

Private Sub CopyMessage()
    Set gwOldMsg =
gwAccount.Mailbox.Messages.Item(gwAccount.Mailbox.Messages.Count)
    Set gwMsgType = gwOldMsg.ClassName
    Set gwNewMsg = gwAccount.WorkFolder.Messages.Add(gwMsgType)
    gwNewMsg.Recipients.Add gwAccount.Owner
    gwNewMsg.FromText = gwOldMsg.FromText
    gwNewMsg.Subject.PlainText = gwOldMsg.Subject.PlainText
    gwNewMsg.Bodytext.Rtf = gwOldMsg.Bodytext.Rtf
    If gwMsgType = "GW.MESSAGE.APPOINTMENT"
        Set gwNewMsg.Startdate = gwOldMsg.Startdate
        Set gwNewMsg.Enddate = gwOldMsg.Enddate
        Set gwNewMsg.Place = gwOldMsg.Place
    End If
    If gwMsgType = "GW.MESSAGE.NOTE"
        Set gwNewMsg.Startdate = gwOldMsg.Startdate
    End If
    If gwMsgType = "GW.MESSAGE.PHONE"
        Set gwNewMsg CallerCompany = gwOldMsg.CallerCompany
        Set gwNewMsg.CallerName = gwOldMsg.CallerName
        Set gwNewMsg.CameToSee = gwOldMsg.CameToSee
        Set gwNewMsg.PhoneNumber = gwOldMsg.CameToSee
        Set gwNewMsg.PleaseCall = gwOldMsg.PleaseCall
        Set gwNewMsg.ReturnedCall = gwOldMsg.ReturnedCall
        Set gwNewMsg.Telephoned = gwOldMsg.Telephoned
        Set gwNewMsg.Urgent = gwOldMsg.Urgent
        Set gwNewMsg.WantsToSee = gwOldMsg.WantsToSee
        Set gwNewMsg.WillCall = gwOldMsg.WillCall
    End If
    If gwMsgType = "GW.MESSAGE.TASK"
        Set gwNewMsg.assigneddate = gwOldMsg.AssignedDate
        Set gwNewMsg.duedate = gwOldMsg.DueDate
        Set gwNewMsg.startdate = gwOldMsg.StartDate
        Set gwNewMsg.taskcategory = gwOldMsg.TaskCategory
        Set gwNewMsg.taskpriority = gwOldMsg.TaskPriority
    End If
```

```

    Set gwSentMsg = gwNewMsg.Send
    gwAccount.Refresh
    Set gwMsgID = gwSentMsg.MessageID
End Sub

```

Example in Delphi:

```

//gwAccount is a valid Account object and the last message is one of the above
message types
procedure CopyMessage;
var gwMsgType,gwMsgID:string;gwSentMsg,gwOldMsg,gwNewMsg,gwMailBox:variant;
begin
    gwOldMsg:=gwAccount.Mailbox.Messages.Item(gwAccount.Mailbox.Messages.Count);
    gwMsgType:=gwOldMsg.ClassName; //find out what type of message this is
    gwNewMsg:=gwAccount.WorkFolder.Messages.Add(gwMsgType); {create a new message
    of that type}
    gwNewMsg.Recipients.Add(gwAccount.Owner); {add the account owner's address
    to the recipients}
    gwNewMsg.FromText:=gwOldMsg.FromText;
    gwNewMsg.Subject.PlainText:=gwOldMsg.Subject.PlainText;
    gwNewMsg.Bodytext.Rtf:=gwOldMsg.Bodytext.Rtf;
    {note that we don't need to process for GW.MESSAGE.MAIL as there are no
    special fields}
    if (CompareText('GW.MESSAGE.APPOINTMENT',Copy(gwMsgType,1,22)) = 0) then
    begin {if appointment or appointment subclass}
        gwNewMsg.Startdate:= gwOldMsg.Startdate;
        gwNewMsg.Enddate:= gwOldMsg.Enddate;
        gwNewMsg.Place:=gwOldMsg.Place;
    end;
    if (CompareText('GW.MESSAGE.NOTE',Copy(messtype,1,15)) = 0) then begin {if
    note}
        gwNewMsg.Startdate:=gwOldMsg.Startdate;
    end;
    if (CompareText('GW.MESSAGE.PHONE',Copy(messtype,1,16)) = 0) then begin {if
    phone message}
        gwNewMsg.CallerCompany:= gwOldMsg.CallerCompany;
        gwNewMsg.CallerName := gwOldMsg.CallerName;
        gwNewMsg.CameToSee := gwOldMsg.CameToSee;
        gwNewMsg.PhoneNumber := gwOldMsg.PhoneNumber;
        gwNewMsg.PleaseCall := gwOldMsg.PleaseCall;
        gwNewMsg.ReturnedCall := gwOldMsg.ReturnedCall;
        gwNewMsg.Telephoned := gwOldMsg.Telephoned;
        gwNewMsg.Urgent := gwOldMsg.Urgent;
        gwNewMsg.WantsToSee := gwOldMsg.WantsToSee;
        gwNewMsg.WillCall := gwOldMsg.WillCall;
    end;
    if (CompareText('GW.MESSAGE.TASK',Copy(messtype,1,15)) = 0) then begin {if
    task}
        gwNewMsg.assigneddate:= gwOldMsg.AssignedDate;
        gwNewMsg.duedate:= gwOldMsg.DueDate;
        gwNewMsg.startdate:= gwOldMsg.StartDate;
        gwNewMsg.taskcategory:= gwOldMsg.TaskCategory;
        gwNewMsg.taskpriority:= gwOldMsg.TaskPriority;
    end;
    gwSentMsg:=gwNewMsg.Send; {send the new message}

```

```

gwAccount.Refresh; {refresh the database}
gwMsgID:=gwSentMsg.MessageID;{get the messageid of the new message - remember
that outgoing and incoming messageid's are different}
end;

```

Attachments

Using the Object API, your code can easily access the attachments of a message (including draft messages) that are already in the GroupWise database. Like many other collection objects, Attachments has a Count property and Item() method. To access a specific attachment, you must call the Item() method with a single Variant parameter.

```
Attachments.Item(Index as Variant)
```

If Index is Long between 1 and Attachments.Count, the Attachment object at the given Index will be returned. Use a for loop if you don't know the order of attachments in the collection. If Index is a string, it should represent the MessageID of the Attachment object you are trying to get. If Index is a Message object, it represents the desired Message object attachment.

The Attachment Object

The key properties of the Attachment object are:

Property	Data Type	Description
DisplayName	String (R/O)	Contains the string that the user would see if the message were viewed in the GroupWise client.
FileName	String (R/O)	If ObjType is egwFile (or any other value than egwMessage), then this property will contain a string that is the filename of the—attached file. Note that this is not related to a GroupWise document there is no object in the Object API that represents this file (which is different than a document reference, which is persistent in the GroupWise database).
Message	Message object (R/O)	If ObjType is egwMessage, then this property will contain a Message object corresponding to the attached message. It is important to note that if the attached message is owned by another user, the ability to manipulate the message will be limited. For example, you may not be able to add the message to a folder in the current account. Nonetheless, you will be able to access the primary properties of the message, such as BodyText. Further, if the message is a DocumentReference object (check the ClassName property to determine this), you can manipulate the document as described above. Note that FileName and Message are mutually exclusive; thus, if one holds data, the other should be empty.

Property	Data Type	Description
ObjType	Enum (R/O)	Determines exactly what the attachment is, given that an attachment can be many different things. The primary types to become familiar with are egwFile (1) which is an attached file, and egwMessage(2) which is an attached GroupWise message.

Referencing Attachments and Attached Message Properties

The following code shows the Attachment.DisplayName property in use. It shows the name of all attachments of the last message in the Mailbox.

Example in VB:

```
'gwAccount is a valid Account object and gwMessage has a valid Attachments
collection

Dim gwMessage As GroupwareTypeLibrary.Message
Dim gwAttachment As GroupwareTypeLibrary.Attachment

Private Sub ShowAttachments()
Set gwMessage =
gwAccount.Mailbox.Messages.Item(gwAccount.Mailbox.Messages.Count)
For i=1 to gwMessage.Attachments.Count
    Set gwAttachment = gwMessage.Attachments.Item(i)
    MsgBox gwAttachment.DisplayName
Next I
End Sub
```

Example in Delphi:

```
//gwAccount is a valid Account object and gwMessage has a valid Attachments
collection

procedure ShowAttachments;
var i:integer;gwMessage,gwAttachment:variant;
begin
gwMessage:=gwAccount.Mailbox.Messages.Item(gwAccount.Mailbox.Messages.Count);
for i := 1 to gwMessage.Attachments.Count do
    begin
        gwAttachment:=gwMessage.Attachments.Item(i);
        ShowMessage(gwAttachment.DisplayName);
    end;
end;
```

This next example will iterate through all of the attachments in the last message in the mailbox, and determine the type of each attachment. If the attachment is a message, it will check to see whether the message is a document reference. If so, it shows the subject of the document reference. If not (and it is thus a “regular” message) then the procedure will show who sent the message. If the attachment is a file, it will show the filename.

Example in VB:

```
'gwAccount is a valid Account object and gwMessage has a valid Attachments
collection

Dim gwMessage As GroupwareTypeLibrary.Message
Dim gwAttachment As GroupwareTypeLibrary.Attachment

Private Sub ProcessAttachments()
Set gwMessage = gwAccount.Mailbox.Messages.Item(gwAccount.Mailbox.
Messages.Count)
For doc=1 to gwMessage.Attachment.Count
  Set gwAttachment = gwMessage.Attachments.Item(doc)
  If gwAttachment.ObjType = 2 Then
    If gwAttachment.ClassName = "GW.MESSAGE.DOCUMENTREFERENCE" Then
      MsgBox "The message subject: " & gwAttachment.Message.Subject
    Else
      MsgBox "The message is from: " & gwAttachment.Message.FromText
    End If
  End If
  If gwAttachment.ObjType = 1 Then
    MsgBox "The filename is: " & gwAttachment.FileName
  End If
Next doc
End If
End Sub
```

Example in Delphi:

```
//gwAccount is a valid Account object and gwMessage has a valid Attachments
collection

procedure ProcessAttachments;
var doc:integer;gwMessage,gwAttachment:variant;
begin
gwMessage:=gwAccount.Mailbox.Messages.Item(gwAccount.Mailbox.Messages.Count);
for doc := 1 to gwMessage.Attachments.Count do
  begin
    gwAttachment:=gwMessage.Attachments.Item(doc);
    if gwAttachment.ObjType = egwMessage then begin
      if gwAttachment.ClassName = "GW.MESSAGE.DOCUMENTREFERENCE"
then
        ShowMessage('The message subject: ' +
gwAttachment.Message.Subject);
      else
        ShowMessage('The message is from: ' +
gwAttachment.Message.FromText);
      end;
    end;
    if gwAttachment.ObjType =egwFile then begin
      ShowMessage('The filename is: ' + gwAttachment.FileName);
    end;
  end;
end;
end;
```

Adding Attachments

Use one of two `Add()` methods of the `Attachments` object collection to create a new `Attachment` object.

Note: Your code will cause an exception if the object that owns the `Attachments` collection you are trying to use is not a draft or personal message.

This first version of `Add()` accepts up to three parameters:

```
Attachments.Add(FileName as String, [ObjType as AttachmentTypeConstants],[DisplayName as String])
```

The first parameter, `FileName`, is required and sets the `Attachment.FileName` property. The second parameter, if it exists, sets the `attachments ObjType` property. The default is `egwFile`. Finally, the third parameter, if it exists, sets the `attachments DisplayName` property. For this to work correctly, `ObjType` of the `Attachment` should not be set to `egwMessage`.

The second version of `Add()` accepts up to two parameters:

```
Attachments.Add(Message as Message, [DisplayName as String])
```

The first parameter, `Message`, represents a `Message` object that you are trying to add to your message. This parameter is required unlike the second which sets the `attachments DisplayName` property as described in the previous method. Unlike the previous `Add()`, for this method to work correctly, `ObjType` should be set to `egwMessage`.

Saving Attachments

Use the `Save()` method to save an attachment where `Attachment.ObjType` is not `egwMessage` to disk.

Attachment.Save(Filename as String)

This method takes a single string parameter called `Filename` which contains the filename you wish to save the attachment as. If you omit a path, the file will be saved in the default location. Note that the `Save()` method is the only way to access an attached file. `GroupWise` does not allow for “in database” access of any files.

The following example iterates through the attachments in a given message using the `Attachments.Item()` method. Attachments will be saved to disk with `Attachment.Save()` and the attachments prior file name. Because there is no path, the attachment will be saved in the default location.

Example in VB:

```
'gwMail is a valid Mail object with attachments

Dim gwAttachment As GroupwareTypeLibrary.Attachment

Private Sub SaveAttachments(gwMail)
If gwMail.Attachments.Count > 0 Then
  For i=1 to gwMail.Attachments.Count
    Set gwAttachment = gwMail.Attachments.Item(i)
    If gwAttachment.ObjType <> egwMessage Then
      gwAttachment.Save gwAttachment.FileName
    End If
  Next i
End If
Next I
End Sub
```

Example in Delphi:

```
//gwMail is a valid Mail object with attachments

procedure SaveAttachments(gwMail:variant);
var i:integer;gwAttachment:variant;
begin
If gwMail.Attachments.Count > 0 then
  begin
    for i:= 1 to gwMail.Attachments.Count do
      begin
        gwAttachment:=gwMail.Attachments.Item(i);
        If gwAttachment.ObjType <> egwMessage then
          gwAttachment.Save(Attachment.FileName);
        end;
      end;
    end;
  end;
end;
```

Deleting an Attachment

Use the Delete() method on an Attachment object to delete an attachment from the objects parent collection.

```
Attachment.Delete()
```

This method requires no parameters and will work on a draft or personal item only.

Recipients

It is important that you correctly understand the role that Recipient related objects play in GroupWise. For information on what the Recipient object is, see Chapter 7, Understanding Address and AddressBook Objects.

Recipients, like many other collections objects, has a Count property and Item() method that allows you to access individual Recipient objects. Simply call the Item() method with a single Index parameter.

```
Recipients.Item(Index as Long)
```

If Index is Long between 1 and Attachments.Count, the Attachment object at the given Index will be returned. Use a for loop if you don't know the order of attachments in the collection.

The Recipient Object

The key properties of the Recipient object are:

Property	Data Type	Description
Address	Address object (R/O)	The address this recipient resolved to. Can be nothing if this recipient is unresolved or resolved as an external address.
DisplayName	String (R/W)	A descriptive name that is displayed to users.
EmailAddress	String (R/W)	The e-mail address used by the system to deliver mail. The format is determined by the EmailType property.
EmailType	String (R/W)	The type of e-mail address. "NGW" indicates an internal GroupWise address. Anything else is an external address. External addresses are submitted to the operating system's default email transport. (In Windows, for example, the transport would be MAPI.)
Resolved	Enum (R/W)	Indicates the resolved status of this recipient. Automatically set to egwNotResolved when the DisplayName, EmailAddress, EmailType, or TargetType properties are changed. Can be manually set only to egwNotResolved. This property is not persistent. It will revert to egwNotResolved whenever the Recipient object is refreshed or freed from memory. [egwNotResolved (0), egwNotFound (1), egwAmbiguous (2), egwResolved (3)]
TargetType	Enum (R/W)	Indicates whether or not the recipient is a primary recipient, carbon copy recipient, or blind copy recipient. [egwTo (0), egwCC (1), egwBC (2)]

Adding Recipients

Four different Add() methods allow you to easily populate the Recipients collection of a Message object before you send.

Use the Add() method below to add a single Address object to the Recipients collection.

```
Recipients.Add(Address as Address, [TargetType as AddressTargetTypeConstants])
```

This method takes up to two parameters and creates an unresolved Recipient object from the Address object which is passed in. The method returns the new Recipient object which it has added to the collection with its Address property set to the Address object that it received. DisplayName, EmailAddress, and EmailType are copied from the Address object into the new object. TargetType, is optional and if omitted, defaults to egwTo.

Use the second form of Add() when you want to add entire collections of Address objects to a Recipients collection.

```
Recipients.Add(Addresses as Addresses, [TargetType as AddressTargetTypeConstants])
```

This method creates an unresolved Recipient for each Address in the Addresses collection that is passed in as the first parameter. In turn, each Recipient object is added to the Recipients collection until all Address objects have been added. Again, if TargetType is omitted, egwTo is assumed. Unlike, the first form of Add(), this method returns nothing.

Use the third form of Add() when you know the email address of the recipient that you would like to add.

```
Recipients.Add(EmailAddress as String, [EmailType as String], [TargetType as AddressTargetTypeConstants])
```

Like the second, this method also creates an unresolved Recipient which it adds to the Recipients collection. An empty string ("") is assumed if EmailType is omitted and egwTo is assumed if TargetType is omitted. The new recipient is returned as soon as the Recipient.Address is set to Nothing and Recipient.DisplayName is set to EmailAddress.

```
Recipients.AddByDisplayName(DisplayName as String, [TargetType as AddressTargetTypeConstants TargetType])
```

This method creates an unresolved Recipient from DisplayName, which it then adds it to the Recipients collection. If DisplayName is an empty string (""), an exception is thrown. If TargetType is omitted, egwTo is assumed. The new Recipient Address property is set to Nothing and its EmailAddress and EmailType are set to an empty string (""). The new Recipient object is then returned.

Resolving Recipients

Use the Resolve() method on either a single Recipient object or on an entire Recipients collection to resolve one or multiple recipients.

```
Recipient.Resolve([ResolveTo as AddressBook])  
Recipients.Resolve([ResolveTo as AddressBook])
```

These methods take as an optional parameter an AddressBook object, which they will use to try and locate the recipient(s) in question. If the Recipient is already resolved (its Resolved property is TRUE), the Resolve operation is considered successful.

Note: If this method is called on a Recipients object collection, it will attempt to resolve each object in the collection. If an error occurs, it leaves that recipient unresolved and proceeds to the next recipient. Check the Resolved property of each Recipient object to determine which ones failed. An exception is thrown if any recipients failed to resolve.

When an entry is found that matches the Recipient, a new Address property is created for the Recipient and its values are copied from the found address book entry. The Resolve operation is considered successful.

If a recipient is not found in an address book and its EmailType is external (such as an Internet address), the Recipients Address property is set to a new Address object with ObjType = egwUser and DisplayName, EmailAddress, and EmailType the same as the Recipient's properties, the Resolve operation is considered successful.

If the recipient is not found in an address book and its EmailType is internal (such as a GroupWise address), the Resolve operation throws an exception.

Deleting Recipients

Use the Delete() method on a Recipient object to remove the recipient from the owning Recipients collection.

```
Recipient.Delete()
```

This method requires no parameters.

Summary

In this chapter you learned how your application can create, delete and access the information stored in various properties of a GroupWise message using the Object API. You learned about various properties and methods of an Appointment, DocumentReference, Mail, Note, PhoneMessage, and Task object, how to add file and messages items to a messages Attachments collection and also how to use the Recipients collection of a Message object when you want to address and send messages. In the next chapter you will learn about Document and Document related objects.

For more information on Messages objects, please check the GroupWise Object API documentation <http://developer.novell.com/ndk/doc.htm>, sample code <http://developer.novell.com/ndk/sample.htm> or visit the Developer Support Forum area at <http://developer-forums.novell.com/category/index.tpt>.

Copyright © 2002 by Novell, Inc. All rights reserved.
No part of this document may be reproduced or transmitted
in any form or by any means, electronic or mechanical,
including photocopying and recording, for any purpose
without the express written permission of Novell.

All product names mentioned are trademarks of
their respective companies or distributors.

Understanding Document and Document Related Objects

Chapter 6

Section 1: GroupWise Object API

The GroupWise Object API provides a complex set of intertwined objects to manage documents. DocumentReference objects which are really just special message objects, point to documents that a user has stored.

A Document object holds default rights information granted to everyone in a DocumentAccessRights object of a DocumentAccessRightsCollection. Rights information for the document's author, a specific user or for a group is also available as is versioning and library information in which the document resides.

A DocumentIterator object is used to access individual document objects from within a Documents collection.

The following topics are discussed in the chapter.

Contents:

- Document References
- Document Libraries
- The Documents Collection
- The Document and DocumentVersions Objects
- Document Events
- Document Rights
- Document Types

Document References

The `DocumentReference` object is a special type of message, primarily because it does not have message text associated with it. Instead, the `DocumentReference` can best be described as a pointer to a BLOB document file in the GroupWise document library. Because the `DocumentReference` is only a pointer, you may not manipulate documents directly through the Object API. Rather, you must save a copy of the document to the local hard disk, and then manipulate the document using whatever tools and APIs are available to edit the document on disk.

The key properties of the `DocumentReference` object are:

Property	Data Type	Description
Document	Document object (R/O)	This property is a reference to the actual Document object as it is represented in the GroupWise library. The Document object is discussed in more detail below.
DocumentLibrary	DocumentLibrary object (R/O)	This property is a reference to a DocumentLibrary type object for the Document at issue. The DocumentLibrary object is not discussed at length here. It is a collection of Document objects, and has an <code>Add()</code> method to add new documents. Further DocumentLibrary does not have a <code>Count</code> property and instead requires iteration through each document, which can be cumbersome.
DocumentVersion	DocumentVersion object (R/O)	This property is a reference to a DocumentVersion type object that holds information about this documents version. The DocumentVersion object is described in more detail below.
RefType	Enum	This is an enumerated property that holds information about the type of document reference. [egwOfficial (0) for the official version, egwCurrent (1) for the current version, and egwSpecific (2) for a specific version]

The `DocumentReference` object has one method of note, called `LocalDelete()`.

```
DocumentReference.LocalDelete()
```

This method takes no parameter, and moves the specific `DocumentReference` object to the trash can without modifying the underlying document.

Let's take another look at the `ProcessAttachments` procedure from the attachments section in Chapter 5: *Understanding Document and Document Related Objects*. In that section, we assumed that the attached message was not a document reference. In reality, however, we must usually test for document references because they have different properties than most other messages. The code below performs this test, and shows how to access the subject of a document reference (which is the subject that users see):

Example in VB:

```
'gwAccount is a valid Account object

Dim gwMessage As GroupwareTypeLibrary.Message
Dim gwAttachment As GroupwareTypeLibrary.Attachment

Private Sub ProcessAttachments()
Set gwMessage =
gwAccount.Mailbox.Messages.Item(gwAccount.Mailbox.Messages.Count)
For doc=1 to gwMessage.Attachments.Count
Set gwAttachment = gwMessages.Attachment.Item(doc)
If gwAttachment.ObjType = egwMessage Then
If gwAttachment.Message.ClassName = "GW.MESSAGE.DOCUMENTREFERENCE" Then
Msgbox "The Document Reference subject is: " &
gwAttachment.Message.Subject.PlainText
Else
Msgbox "The message is from: " & gwAttachment.Message.FromText
End If
End If
If gwAttachment.ObjType = egwFile Then
Msgbox "The Filename is: " & gwAttachment.Filename
End If
Next doc
End Sub
```

Example in Delphi:

```
//gwAccount is a valid Account object

procedure ProcessAttachments;
var doc:integer; gwMessage,gwAttachment:variant;
begin
gwMessage:=gwAccount.Mailbox.Messages.Item(gwAccount.Mailbox.Messages.Count);
for doc := 1 to gwMessage.Attachments.Count do
begin
gwAttachment:=gwMessage.Attachments.Item(doc);
if gwAttachment.ObjType = egwMessage then begin
    {now we must test to see whether we have a message or a document
    reference, because the properties are different}
if gwAttachment.Message.ClassName='GW.MESSAGE.DOCUMENTREFERENCE' then
ShowMessage('The Document Reference subject is: ' +
    gwAttachment.Message.Subject.PlainText)
else
ShowMessage('The message is from: ' +
    gwAttachment.Message.FromText);
end;
end;
```

```
if gwAttachment.ObjType =egwFile then begin
ShowMessage('The Filename is: '+ gwAttachment.FileName);
end;
end;
end;
```

Document Libraries

Document libraries hold the documents that GroupWise users create and edit. You can access `DocumentLibrary` objects from the following objects:

```
Account.DocumentLibraries
Account.DefaultDocumentLibrary
Document.DocumentLibrary
DocumentReference.DocumentLibrary
DocumentVersion.DocumentLibrary
The DocumentLibraries Object
```

The `DocumentLibraries` object is a collection of `DocumentLibrary` objects. Like most other collections, `DocumentLibraries` has a `Count` property and `Item()` method.

This `Count` property is an integer that holds the number of libraries in the collection.

Use the `Item()` method to obtain a specific library object from the collection.

```
DocumentLibraries.Item(Index as Variant)
```

`Item()` takes as single parameter called `Index`, which is a variant. You can make `Index` an integer between 1 and `DocumentLibraries.Count`, and the `DocumentLibrary` object corresponding to that position will be retrieved. You may alternatively make `Index` a string. Of course, not every string will work; rather, the string you pass must be a valid `LibraryID` to one of the `DocumentLibrary` objects in the collection. If `Index` is a valid `LibraryID`, the `DocumentLibrary` object with that `LibraryID` will be returned.

Example in VB:

```
'gwAccount is a valid Account object

Dim gwDocLibrary As GroupwareTypeLibrary.Documentlibrary

Private Sub GetFirstDocumentLibrary ()
If gwAccount.DocumentLibraries.Count > 0 Then
Set gwDocLibrary = gwAccount.DocumentLibraries.Item(1)
Msgbox "gwDocLibrary now points to a DocumentLibrary object"
End If
End Sub
```

Example in Delphi:

```
//gwAccount is a valid Account object

procedure GetFirstDocumentLibrary;
var gwDocLibrary:variant;
begin

If gwAccount.DocumentLibraries.Count > 0 then
gwDocLibrary:=gwAccount.DocumentLibraries.Item(1);
    {gwDocLibrary now points to a DocumentLibrary object}
end;
```

The DocumentLibrary Object

The key properties of the DocumentLibrary are:

Name	Data Type	Description
CurrentArchive Bytes	Integer (R/O)	The number of bytes that the archive takes on disk
CurrentArchive Location	String (R/O)	The full path and name to the location of the documents archive on disk
Description	String (R/O)	Description of the library
Documents	Documents object (R/O)	Holds all of the documents in the library
DocumentTypes	Document Types object (R/O)	The collection of document types that are associated with the library. Note that document type here means not Word v. WordPerfect, but rather the document types defined in the GroupWise administration for archiving and automatic deletion. The DocumentLibrary.DocumentTypes property is a collection of all the available document types for documents.
Field Definitions	Field Definitionsobject (R/O)	The collection of field definitions for the library.
LibraryID	String (R/O)	The LibraryID for the library. Typically, this will be Domain.PostOffice.LibraryName
MaxArchive Bytes	Integer (R/O)	The maximum size of the archive on disk, as set by the GroupWise administrator
Name	String (R/O)	Name of the library
NextDocument Number	Integer (R/O)	The next document number to be assigned in the library. This does not work in GroupWise remote. You should use DocumentLibrary.Refresh() before accessing this property to make sure that you have the latest information from the GroupWise database

Name	Data Type	Description
Starting Version Number	Integer (R/O)	The version number assigned to new documents by default. This value is almost always 1

Let's look at some property values for the first collection of Documents in the library.

Example in VB:

```
'gwAccount is a valid Account object

Dim gwDocLibrary As GroupwareTypeLibrary.DocumentLibrary

Private Sub GetFirstDocumentLibrary ()
If gwAccount.DocumentLibraries.Count > 0 Then
Set gwDocLibrary = gwAccount.DocumentLibraries.Item(1)
Msgbox "Name: " & gwDocLibrary.Name
Msgbox "Descriptio: " & gwDocLibrary.Description
Msgbox "LibraryID: " & gwDocLibrary.LibraryID
End If
End Sub
```

Example in Delphi:

```
//gwAccount is a valid Account object

procedure GetFirstDocumentLibrary;
var gwDocLibrary:variant;
begin
If gwAccount.DocumentLibraries.Count > 0 then
begin
FirstLibrary:=gwAccount.DocumentLibraries.Item(1);
ShowMessage('Name: ' + gwDocLibrary.Name);
ShowMessage('Description: ' + gwDocLibrary.Description);
ShowMessage('LibraryID: ' + gwDocLibrary.LibraryID);
end;
end;
```

The DocumentLibrary object also has a variety of methods associated with it.

Locating a Document. If you know the document number, you can use `GetDocument ()` to obtain a reference to the document.

```
DocumentLibrary.GetDocument(DocumentNumber as Integer)
```

`DocumentNumber` is an integer that contains a valid document number. The method returns a reference to the Document object associated with `DocumentNumber`.

The following code will attempt to obtain a random document in the user's mailbox. If the user is a librarian, this should return a document every time. If not, then it will only return a document if the user has a reference to the document in his or her mailbox. The code will test to see whether a document was returned. If you want this code to always retrieve a document, set docnum equal to gwDocLibrary.NextDocumentNumber-1, which should be the last document in the user's database.

Example in VB:

```
'gwAccount is a valid Account object

Dim gwDocLibrary As GroupwareTypeLibrary.DocumentLibrary
Dim gwDocument As GroupwareTypeLibrary.Document

Private Sub GetRandomDoc()
If gwAccount.DocumentLibraries.Count > 0 Then
Set gwDocLibrary = gwAccount.DocumentLibraries.Item(1)

`Randomize here
End If
End Sub
```

Example in Delphi:

```
//gwAccount is a valid Account object

procedure GetRandomDoc;
var gwDocLibrary,gwDocument:variant;seed,gwDocNum:integer;
begin
If gwAccount.DocumentLibraries.Count > 0 then
begin
gwDocLibrary:=gwAccount.DocumentLibraries.Item(1);
Randomize;
seed:=FirstLibrary.NextDocumentNumber-1;
docnum:=Trunc(Random(seed));
try
gwDocument:=gwDocLibrary.GetDocument(gwDocNum);
except ShowMessage('Cannot access document number: ' +inttostr(gwDocNum)); end;
if varisempty(gwDocument) then
ShowMessage('Cannot access document number: ' +inttostr(gwDocNum));
end;
end;
```

Getting Document Event Information. Use GetDocumentVersionEvents () to find out what has happened to a document.

```
DocumentLibrary.GetDocumentVersionEvents(DocNumber as Long)
```

DocNumber is an integer that contains a valid document number. The method returns a reference to a DocumentVersionEvents object, from which you can find out what has occurred with the document associated with DocNumber.

Example in VB:

```
'gwAccount is a valid Account object

Dim gwDocLibrary As GroupwareTypeLibrary.Documentlibrary
Dim gwDocumentEvents As GroupwareTypeLibrary.Document

Private Sub GetDocEvents()
If gwAccount.DocumentLibraries.Count > 0 Then
Set gwDocNum = gwAccount.NextDocumentNumber - 1
On Error goto dfadasfda
Set gwDocumentEvents = gwDocLibrary.GetDocumentVersionEvents(gwDocNum)
End If
End Sub
```

Example in Delphi:

```
//gwAccount is a valid Account object

procedure GetDocEvents;
var gwDocLibrary,gwDocumentEvents:variant;seed,docnum:integer;
begin
If gwAccount.DocumentLibraries.Count > 0 then
begin
gwDocLibrary:=gwAccount.DocumentLibraries.Item(1);
docnum:=gwDocLibrary.NextDocumentNumber-1;
try
gwDocumentEvents:=gwDocLibrary.GetDocumentVersionEvents(docnum);
except end;
if varisempty(gwDocumentEvents) then
ShowMessage('Cannot access document number: ' +inttostr(docnum));
end;
end;
```

Archiving Documents. Use IncrementArchiveLocation() to begin archiving documents in the next available directory.

```
DocumentLibrary.IncrementArchiveLocation ()
```

This method is not available in GroupWise Remote.

The Documents Collection

The `Documents` object is a collection of `Document` objects. You obtain a reference to `Documents` by referencing the `DocumentLibrary` object. The `DocumentLibrary.Documents` object will contain all of the documents in that library.

There are two primary uses for a `Documents` object. First, it allows you to iterate through all documents. Second, it allows you to add a new document to the library.

Iterating through Documents

Use the `Documents.CreateDocumentIterator()` method to allow for iteration through each document in the library. Because `Documents` is a large collection, like `AllMessages`, there is no `Item()` method to individually index documents.

```
Documents.CreateDocumentIterator()
```

This token returns a reference to a `DocumentIterator` object. From there, you should call the following methods of `DocumentIterator`.

Method	Description
<code>Clone()</code>	This method will create a copy of the <code>DocumentIterator</code> with the iterator at the same position as the cloned object. This makes it possible to keep track of multiple iteration positions, and go back if necessary.
<code>Next()</code>	This takes no parameters, and returns the next <code>Document</code> object in the collection.
<code>Reset()</code>	This takes no parameters, and returns the iterator to before the first document so that calling <code>Next()</code> will return the first document.
<code>Skip(NumDocuments)</code>	This takes one long integer parameter called <code>NumDocuments</code> . <code>Skip(NumDocuments)</code> will cause the iterator to skip over <code>NumDocuments</code> messages, placing the iterator just before the document following the skipped documents so you may then call a <code>Next()</code> method. This is mostly useless for several reasons. First, <code>NumDocuments</code> must be positive, so you may not move backward. Second, the documents are not in any particular order, so you do not know what you are skipping. Third, there is no end of record checking, in case <code>NumDocuments</code> will move the iterator past the last document.

The following procedure shows each of the above methods in action.

Example in VB:

```
'gwAccount is a valid Account object  
  
Dim gwDocLibrary As GroupwareTypeLibrary.DocumentLibrary  
Dim gwDocument As GroupwareTypeLibrary.Document
```

```

Dim gwDocIterator As GroupwareTypeLibrary.DocumentIterator
Dim gwDocIterator2 As GroupwareTypeLibrary.DocumentIterator

Private Sub IterateDocs()
Set gwDocLibrary = gwAccount.DefaultDocumentLibrary
Set gwDocIterator = gwDocLibrary.Documents.CreateDocumentIterator
Set gwDocIterator = gwIterator.Next
Msgbox "Subject: " & cStr(gwDocument.DocumentNumber) & " " & gwDocument.Subject
gwDocIterator.Skip 2
gwDocument.gwDocIterator.Next
Msgbox "Skipped two, so now on forth doc: " & " " &
cStr(gwDocument.DocumentNumber) & " " & gwDocument.Subject
Set gwDocIterator2 = gwDocIterator.Clone
gwDocIterator.Reset
Set gwDocument = gwDocIterator.Next
Msgbox "gwDocIterator, back to the beginning: " &
cStr(gwDocument.DocumentNumber) & " " & gwDocument.Subject
Set gwDocument = gwDocIterator2.Next
Msgbox "gwDocIterator2, next document: " & cStr(gwDocument.DocumentNumber) & " "
& gwDocument.Subject
End If
End Sub

```

Example in Delphi:

```

//gwAccount is a valid Account object

procedure IterateDocs;
var gwDocLibrary,gwDocIterator,gwDocIterator2,gwDocument:variant;
begin
gwDocLibrary:=gwAccount.DefaultDocumentLibrary;
gwDocIterator:=gwDocLibrary.Documents.CreateDocumentIterator;
gwDocument:=gwDocIterator.Next;
ShowMessage('Subject: ' + inttostr(gwDocument.DocumentNumber) + ' ' +
gwDocument.Subject);
gwDocIterator.Skip(2);
gwDocument:=gwDocIterator.Next;
ShowMessage('Skipped two, so now on fourth doc: ' +
inttostr(gwDocument.DocumentNumber) + ' ' + gwDocument.Subject);
gwDocIterator2:=gwDocIterator.Clone;
gwDocIterator.Reset;
gwDocument:=gwDocIterator.Next;
ShowMessage('gwDocIterator, back to the beginning: ' +
inttostr(gwDocument.DocumentNumber) + ' ' + gwDocument.Subject);
gwDocument:=gwDocIterator2.Next;
ShowMessage('gwDocIterator2, next document: ' +
inttostr(gwDocument.DocumentNumber) + ' ' + gwDocument.Subject);
end;

```

Adding New Documents

Use one of two `Documents.Add()` to create a new document in the documents collection. These methods are more like import procedures, because they require a filename to create the new document.

```
Document.Add(Filename as String,[DocumentType as Variant])
Document.AddEx(Filename as String,[DocumentType as Variant],[DocRefFolder as
Folder])
```

Parameter	Data Type	Description
Filename	String	This is the full path and name of the file on disk that you want to use to create the new document.
DocumentType	Variant	This parameter may be either a string that is the name of the DocumentType for the document, or it may be the actual DocumentType object. The DocumentType object is discussed briefly below in the ** section.
DocRefFolder	Folder object	This is a Folder object containing the folder in which to place the new document reference for the document

Both Add () and AddEx () return a Document reference for the newly created document. From there, you can set the other Document properties. AddEx () adds the document reference in DocRefFolder, while Add () does not create a document reference in any folder.

The following procedure will add the c:\config.sys file to the GroupWise library, and create a reference in the mailbox.

Example in VB:

```
'gwAccount is a valid Account object

Dim gwDocument As GroupwareTypeLibrary.Document

Private Sub AddDocument()
Set gwDocument =
gwAccount.DefaultDocumentLibrary.Documents.AddEx("c:\config.sys", , gwAccount.Mailbox)
Msgbox "New Document Number: " & cStr(gwDocument.DocumentNumber)
End Sub
```

Example in Delphi:

```
//gwAccount is a valid Account object

procedure AddDocument;
var gwDocument:variant;
begin
gwDocument:=gwAccount.DefaultDocumentLibrary.Documents.AddEx('c:\config.sys',
{leave second parameter blank for default},gwAccount.Mailbox);
ShowMessage('New Document Number: ' + inttostr(gwDocument.DocumentNumber));
end;
```

The Document and DocumentVersions Objects

The `Document` object is a collection of `DocumentVersion` objects, plus some additional information. Each version of each file in the GroupWise library has an associated `DocumentVersion`. You cannot access and file in the library unless you access that file's `DocumentVersion`. The easiest way to access your `DocumentVersion` of choice is to access the `DocumentReference.DocumentVersion` property. This becomes problematic, however, when the `DocumentReference.RefType` is `egwCurrent`, yet you want a specific version, or vice versa. To access the current version of a document, you can access `DocumentReference.Document.CurrentVersion` property. If you need to access all document versions in order to select one, access the `Document.DocumentVersions` object.

The Document Object

The `Document` object encapsulates all versions of a given document. It has several important properties:

Property	Data Type	Description
Subject	String	This string holds the subject description of the document.
AdditionalRights	DocumentAccess Rights object	This collection contains information about the sharing rights of the document for the particular account that is logged into the Object API.
Author	Address object	This is an Address object containing the address information for the author of this document.
AuthorCreateRights	DocumentAccessRights object (R/O)	Access rights granted to the document's author. If running GroupWise Remote, this property is empty.
CreationDate	Date	This is a date property containing the creation date of the document.
Creator	Address object (R/O)	This is an Address object containing the address information for the creator of this document.
CurrentVersion	DocumentVersion object (R/O)	This is the current version of the document in the form of a <code>DocumentVersion</code> object.
DefaultRights	DocumentAccess Rights object (R/O)	This collection contains the sharing rights granted to everyone.
DocumentLibrary	DocumentLibrary object (R/O)	The library that contains the document

Property	Data Type	Description
DocumentNumber	DocumentVersion object (R/O)	This is an integer that contains the document number of the document.
DocumentType	DocumentType object (R/W)	This property contains information in a DocumentType object, which holds information about the type of document in the library.
Fields	Fields object (R/O)	The collection of custom fields for the document
OfficialVersion	DocumentVersion object (R/O)	This is the official version of the document in the form of a DocumentVersion object.
Subject	String (R/W)	Descriptive name or subject of the document.

In addition to the Document properties described above, there are also a few methods that you should know about. The first is `Delete()`.

```
Document.Delete()
```

This method takes no parameters, and deletes the document and all versions of it from GroupWise. Be careful with this one! We will not even give sample code lest we accidentally delete an important document.

The next method is `GetVersion()`.

```
Document.GetVersion(VersionNumber as Long)
```

This method is another way to access a version, if you know the version number you want. The single parameter for this method is `VersionNumber`, a `Long` representation of a version number. The method returns the appropriate `DocumentVersion` object.

Finally, you may call the `SetDefaults()` method.

```
Document.SetDefaults()
```

This is a useful method (that takes no parameters) if you programmatically create documents. This sets the document properties to be the default values set by the system administrator.

The DocumentVersions Object

The `DocumentVersions` object is a collection of `DocumentVersion` objects for a given `Document`. `DocumentVersions.Count` is an integer property that contains the number of versions associated with a given `Document`. You can iterate through all document versions by using the `DocumentVersions.Item()` method which, like other collection objects in the Object API, takes an integer `Index` parameter between 1 and `Count`, and returns a `DocumentVersion` associated at the `Index` position. You can also create a new version by calling the `DocumentVersions.Add()` method, which takes no parameters and returns the new `DocumentVersion` object.

The DocumentVersion Object

Key properties of the `DocumentVersion` object are:

Property	Data Type	Description
Archived	Boolean (R/O)	TRUE if the document version is archived.
CheckedOut	Boolean (R/O)	TRUE if the document is checked out.
CreationDate	Date	This date property holds the date this version was created.
Creator	Address object (R/O)	This Address property holds the address information for the user who created this version.
Description	String	This is a string property that holds the description associated with this version.
Document	Document object (R/O)	Document from which this version originated.
DocumentLibrary	DocumentLibrary object (R/O)	Library where this version is stored.
DocumentVersion-Events	DocumentVersion Events object (R/O)	This is a collection object that essentially holds all of the prior actions taken on this version. You can iterate through the <code>DocumentVersionEvent</code> objects in the collection by using the <code>Count</code> property and the <code>Item()</code> method with a valid integer <code>Index</code> parameter.
InUse	Boolean (R/O)	TRUE if the document is in use.
ODMADocumentID	String (R/O)	This is the ODMA document identification string to be used with ODMA function calls.
OriginalFileType	String (R/W))	This is the file extension (such as >WPD= or >DOC=) for the document. You can use this to launch appropriate editing and viewing applications.
RetrievalDate	Date (R/O)	Date and time when the version was last retrieved by an application.

Property	Data Type	Description
Retriever	Address object (R/O)	The user who last retrieved the document version.
StagedFilename	String (R/O)	Holds the local location of a retrieved document.
VersionNumber	Long (R/O)	The version number of this document version.

The `DocumentVersion` object has the following key methods.

Saving the DocumentVersion to a Disk File. A document version in the library does little good if you (and your users) are unable to actually edit the files stored within the GroupWise store. The Object API, through the `DocumentVersion` object, provides three methods to transfer a document to disk for actual editing.

Use `CopyOut()` to save a copy of the document file to disk. Use the `CheckOut()` method check a document version out of the library. Use `Retrieve()` to retrieve the document version from the library, and set the “in use” flag to true.

```
DocumentVersion.CopyOut(Filename as String)
DocumentVersion.Preview(Filename as String)
DocumentVersion.CheckOut([Filename as String])
DocumentVersion.Retrieve(Filename as String)
```

Parameter	Data Type	Description
Filename	String	Full path to where you want the file stored

With `CopyOut()`, you can always obtain the latest version of the file, even if it is checked out or in use. Remember, however, that while the document is checked out or in use, you may not be getting the latest version of the document.

`Preview()` is just like `CopyOut()`, except for a couple of features. First, you must provide a filename, because `Preview()` will not assume that you want to use the default naming scheme; rather, the method was designed with the expectation that you will copy the file to a temp directory. Second, the `Preview()` method will be logged differently by GroupWise. That is, it will be treated as if you viewed the document (rather than saved the document) from the GroupWise client.

With `CheckOut()`, if `Filename` is not present, the checked out flag is set, but no file is copied. To obtain access to the file in that case, you will have to use the `CopyOut()` method. If the document is already checked out – whether or not you specified a filename – you will receive an error.

The `Retrieve()` method will generate an error if the document version is already in use or checked out.

Putting the Disk File Back Into the GroupWise Message Store. Once your user has edited the disk file associated with a document, you may want to put it back into the GroupWise message store. The Object API provides methods to do this as well.

Use the `CheckIn()` method to check in a file that has been checked out. Use `EndRetrieve()` to end a retrieve session and update the file contents in the library.

```
DocumentVersion.CheckIn([Filename],[StatusChange])
DocumentVersion.EndRetrieve([Filename],[StatusChange])
DocumentVersion.EndPreview([Filename])
DocumentVersion.RemoteEndRetrieve([Filename],[StatusChange])
```

Parameter	Data Type	Description
Filename	String	The full path to the document file you wish to check in. You may obtain the current checked out filename from <code>DocumentVersion.StagedFilename</code>
StatusChange	Enum	This tells GroupWise how to treat the document version after completion of the operation. Set it equal to <code>egwDoCheckIn/egwDoEndRetrieve (0)</code> if you want to reset the "checked out"/"retrieved" flag or <code>egwLeaveCheckedOut/egwLeaveInUse (1)</code> if you simply want to update the contents in the GroupWise library, but want to keep working on the document. If you omit the parameter, the OAPI will assume a check in or end retrieve.

With `CheckIn()`, if `Filename` is omitted, then the document version is "checked in" with no change in content. If the document version is not checked out when you call this method, you will receive an error.

With `EndRetrieve()`, if `Filename` is omitted, then the document version is marked as no longer "in use" but with no change in content. You will generate an error if the document version has not been retrieved. If the document was retrieved on a remote version, you should use `RemoteEndRetrieve()`.

You should call `EndPreview()` to log the end of the preview action in GroupWise. If you include the filename, the Object API will delete the file. If you do not include the filename, the method will attempt to delete the file in `DocumentVersion.StagedFilename`. This will generate an error if the document has not been put into preview mode with a `Preview()` call.

Deleting Document Versions. If you want to delete a version, use the `Delete()` method.

```
DocumentVersion.Delete()
```

This method will delete all copies of this version from the library. Be careful before calling this method.

Document Events

The DocumentVersionEvents Object

A DocumentVersionEvents, like many other collections objects, has a Count property and Item() method that allows you to access individual DocumentVersionEvents objects. Simply call the Item() method with a single Index parameter.

```
DocumentVersionEvents.Item(Index as Variant)
```

If Index is Long between 1 and DocumentVersionEvents.Count, the DocumentVersionEvents object at the given Index will be returned. Use a for loop if you don't know the order of objects in the collection. If Index is a string, it should represent a Long value.

The DocumentVersionEvent Object

The key properties of the DocumentVersionEvent object are:

Name	Data Type	Description
CreationDate	Boolean (R/O)	TRUE if the user can modify access rights of the owning document. This property cannot be set to TRUE if RevokeAllRights is also TRUE.
Documents	Documents (R/O)	If TRUE, the document user is denied all rights to the document. ModifySecurity (which must be FALSE), SecurityCurrentVersions, SecurityOfficialVersion, SecurityOtherVersions have no meaning and attempting to write to any of these properties will generate an error.
DocumentLibrary	DocumentLibrary (R/O)	Access rights for the documents current version.
DocumentVersion	DocumentVersion (R/O)	Access rights for the documents official version.
EventType	Enum (R/O)	Access rights for the documents unofficial, non current version. For a full list of available EventTypes, see the official GroupWise Object API Documentation.
Filename	String (R/O)	The user or group for whom additional rights are set. May be Nothing if the DocumentAccessRights object is obtained from a documents AuthorCreatorRights or DefaultRights property.

This object does not have any methods.

Document Rights

The DocumentAccessRights Object

DocumentAccessRightsCollections, like many other collections objects, has a Count property and Item() method that allows you to access individual DocumentAccessRights objects. Simply call the Item() method with a single Index parameter.

```
DocumentAccessRightsCollections.Item(Index as Long)
```

If Index is Long between 1 and DocumentAccessRightsCollections.Count, the DocumentAccessRights object at the given Index will be returned. Use a for loop if you don't know the order of objects in the collection.

The DocumentAccessRights Object

The key properties of the DocumentAccessRights object are:

Name	Data Type	Description
ModifySecurity	Boolean (R/W)	TRUE if the user can modify access rights of the owning document. This property cannot be set to TRUE if RevokeAllRights is also TRUE.
RevokeAllRights	Boolean (R/W)	If TRUE, the document user is denied all rights to the document. ModifySecurity (which must be FALSE), SecurityCurrentVersions, SecurityOfficialVersion, SecurityOtherVersions have no meaning and attempting to write to any of these properties will generate an error.
SecurityCurrentVersions	DocumentRights object (R/O)	Access rights for the documents current version.
SecurityOfficialVersion	DocumentRights object (R/O)	Access rights for the documents official version.
SecurityOtherVersions	DocumentRights object (R/O)	Access rights for the documents unofficial, non current version.
User	Address (R/O)	The user or group for whom additional rights are set. May be Nothing if the DocumentAccessRights object is obtained from a documents AuthorCreatorRights or DefaultRights property.

DocumentAccessRights has one method, Delete() which is used to delete a DocumentAccessRights object.

The DocumentRights Object

GroupWise uses the DocumentRights object to define and set access rights for a document that are added to a user's base level access.

The key properties of the DocumentAccessRights object are:

Name	Data Type	Description
AllowDelete	Boolean (R/W)	TRUE if the user can delete the document version.
AllowEdit	Boolean (R/W)	TRUE if the user can modify the document version.
AllowShare	Boolean (R/W)	TRUE if the user can share the document version with others. DocumentVersions are shared by adding them to a shared folder, see Chapter 3: <i>Understanding Folder and Trash Related Objects</i> , and by forwarding them as attachments to a message, see Chapter 5: <i>Understanding Message and Message Related Objects</i> .
AllowView	Boolean (R/W)	TRUE if the user can view the document version.
BitMask	Long (R/W)	The bit mask contains one bit for each Boolean value.

This object does not have any methods.

Document Types

The DocumentTypes Object

DocumentTypes, like many other collections objects, have a Count property and Item() method that allows you to access individual DocumentTypes objects. Simply call the Item() method with a single Index parameter.

```
DocumentTypes.Item(Index as Variant)
```

If Index is Long between 1 and DocumentTypes.Count, the DocumentTypes object at the given Index will be returned. Use a for loop if you don't know the order of objects in the collection. If Index is a string, it should represent a Long value.

The DocumentType Object

The key properties of the DocumentType object are:

Name	Data Type	Description
DisposalMethod	Enum (R/O)	Identifies the method for disposing of the document. [egwDisposeByArchive (1), egwDisposeByDelete (2)]
DocumentLife	Long (R/O)	Number of days following the last modification after which the document will be disposed of.
MaxVersionCount	Long (R/O)	Maximum number of versions for documents of this document type.
Name	String (R/O)	Name of the document type.

This object does not have any methods.

Summary

In this chapter we discussed different Document Related objects. Chapter 7 will discuss different Address objects.

For more information on Documents, please check the GroupWise Object API documentation <http://developer.novell.com/ndk/doc.htm>, sample code <http://developer.novell.com/ndk/sample.htm> or visit the Developer Support Forum area at <http://developer-forums.novell.com/category/index.tpt>.

Copyright © 2002 by Novell, Inc. All rights reserved.
No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, for any purpose without the express written permission of Novell.

All product names mentioned are trademarks of their respective companies or distributors.

Understanding Address and AddressBook Objects

Chapter 7

Section 1: GroupWise Object API

Having already been introduced to Address related objects in Chapter 5, this chapter will build upon what you already know as you are introduced to Address, AddressBookEntry, and GroupMember objects and the respective collection object for each.

You'll also learn about AddressBook objects, how to get and set entries and even create your own books.

The following topics are discussed in this chapter.

Contents:

- Manipulating Addresses and Address Books
- The Different Types of Addresses
- Address Books
- Summary

Manipulating Addresses and Address Books

The `Address` object is relatively simple. It consists of 6 read-only properties and no methods. It is, however returned by 16 properties of other objects and may (or must) be used as arguments in 12 methods of other objects. In this section we will go over this important object. More importantly, we will compare and contrast it to 3 similar objects with which `Address` is sometimes confused: `Recipient`, `AddressBookEntry`, and `GroupMember`.

In short, the properties of an `Address` object uniquely identify a `User`, `Group`, `Resource`, `Company`, or may be `Unknown`.

Property	Data Type	Description
<code>Application</code>	Application object (R/O)	The Application object.
<code>DisplayName</code>	String (R/O)	Descriptive name displayed to users.
<code>EmailAddress</code>	String (R/O)	E-Mail Addresss to deliver mail to. Format varies by type.
<code>EmailType</code>	String (R/O)	"NGW" is used for an internal GroupWise address. Any other string means an outside address. With GroupWise 5.5 internet addressing, you may also use "NGW" for internet addresses.
<code>ObjType</code>	Enum (R/O)	<code>egwUnknown</code> , <code>egwUser</code> , <code>egwCompany</code> , <code>egwResource</code> , <code>egwGroup</code> .
<code>Parent</code>	Various objects (R/O)	The object that owns this address.

An `Address` object is returned by the following properties:

<code>Account.Owner</code>	<code>DocumentVersion.Retriever</code>
<code>AccountRights.Address</code>	<code>DocumentVersionEvent.User</code>
<code>AddressBookRights.Address</code>	<code>Folder.Owner</code>
<code>BusySearchElement.Address</code>	<code>FolderRights.Address</code>
<code>Document.Author</code>	<code>Message.Sender</code>
<code>Document.Creator</code>	<code>Message.Sharer</code>
<code>DocumentAccessRights.User</code>	<code>Recipient.Address</code>
<code>DocumentVersion.Creator</code>	<code>TimeBlock.From</code>

The `Address` object may (or sometimes must) be used in at least one syntax of the following methods:

<code>Account.Proxy</code>	<code>Addresses.Remove</code>
<code>Application.Proxy</code>	<code>AccountRightsCollection.Add</code>
<code>AddressBookEntries.Add</code>	<code>DocumentAccessRightsCollection.Add</code>
<code>AddressBookEntries.Item</code>	<code>Folder.ChangeOwner</code>
<code>Addresses.Add</code>	<code>FolderRightsCollection.Add</code>
<code>Addresses.Item</code>	<code>Recipients.Add</code>

The Different Types of Addresses

So what are the differences between `Address`, `Recipient`, `AddressBookEntry`, and `GroupMember`?

Address

The `Address` object is a resolved version of the information necessary for GroupWise to send a message or identify a resource. A resolved version in this sense means that the address information has been found to reside in an address book, or has been verified to be an external address (see the discussion of resolve under the `Recipient` object discussed next.) It is read only. There are no methods for the `Address` object.

Recipient

A `Recipient` object represents initially unresolved information necessary to send a message. It has a “Resolve” method to attempt to resolve the recipient, and a “Resolved” property that indicates the state of this resolution (egwNotResolved, egwNotFound, egwAmbiguous, egwResolved).

The rules for resolving a recipient are:

GroupWise searches its address books in the following order (unless otherwise specified):

1. Frequent Contacts
2. Personal address books
3. System address book

If it can find a match, it is considered Resolved. It will create an Address object from the information provided. This Address object is listed in the “Address” property of the Recipient object. If GroupWise cannot find a match, then GroupWise determines if it is an internal recipient by checking the value of the EmailType property equal to the string “NGW”. If the address is internal, GroupWise sets the Resolved property to egwNotFound or egwAmbiguous.

Note: This search order of address books can lead to problems of personal address books containing entries of users who have been deleted, or migrated to other post offices. Also, groups might contain users in address books that have already been searched. A useful workaround is to always specify the System address book as the book to resolve to. To access the system address book, use the Account.SystemAddressBook property.

The Recipient object also has a property called “TargetType” of an enumeration type which indicates whether this recipient is a primary recipient (egwTo), carbon copy recipient (egwCC), or blind copy recipient (egwBC).

Other property values of the Recipient object are duplicated in the Address object, such as DisplayName, EmailAddress, EmailType, Parent, and Application. Unlike the Address object, the first three of these properties are read/write. Should some of these values be changed, the “Resolved” property will automatically be reset to egwNotResolved – since no verification has yet been made the new property can be resolved.

The Recipient object also has a method called “Delete” which will allow this Recipient object to be deleted.

AddressBookEntry and GroupMember

An AddressBookEntry object is a subtype of the Address object, and therefore contains all the basic information of the Address object. It can be used as an argument of any method that requires an Address object.

In addition to the type of information that the AddressBookEntry object inherits from the Address object, the AddressBookEntry object also contains additional information. One of these important additional properties is called the “Fields” property. It consists of a collection of “Field” objects, each of which contains additional information about an address entry such as Phone number and Department. There are many system defined fields as well as custom fields that can be created (see Chapter 8: *Understanding Field and Field Related Objects*).

The AddressBookEntry object also contains a “Members” property that is a GroupMembers object collection. This property is valid when the “ObjType” of the AddressBookEntry indicates that it is a “group” (egwGroup). This property allows a particular address book entry to consist of a group of individual entries characterized by some group name. Because the GroupMember object is a subset of the AddressBookEntry object (which in turn is a subset of the Address object), this allows the Members property to consist of a list of AddressBookEntries themselves. Thus, the Members object allows a hierarchy or nesting of groups.

The “Revision Number” object contains the number of times the object is modified, and MasterRevisionNumber contains information about the revision last downloaded to GroupWise Remote. The only unique property of the GroupMember object is called “TargetType”, which allows the user to specify the type of copy (egwTo, egwCC, egwBC) each user in a group gets.

While the documentation indicates many AddressBookEntry and GroupMember properties are read/write, this is only true for Personal address books. The System address book cannot be modified through Object API. The GroupWise Admin Object API (see Chapter 24) must be used to make changes to the System address book.

Both the AddressBookEntry object and the GroupMembers object have a “Delete” method which allows that entry to be deleted from the address book, and a “Refresh” method which allows GroupWise to reread property values from the message database.

Note: For most methods that accept an Address object as a parameter, an AddressBookEntry object can be used instead.

The following table lists the common properties of the four different objects discussed above.

Property	Type	Address object	Recipient object where Message.BoxType = egwDraft	AddressBookEntry, GroupMember objects
Address	Address object		R/O - Nothing if not resolved or resolved to an external address.	
Application	Application object	R/O	R/O	R/O
DisplayName	String	R/O	R/W	R/W
EmailAddress	String	R/O	R/W	R/W
EmailType	String	R/O	R/W	R/W
Fields	Fields collection			R/O (collection). Individual Field members of the collection are R/W. Field members can be added with the Fields.Add() method or removed with Field.Delete().
MasterRevisionNumber RevisionNumber	Long			R/O

Property	Type	Address object	Recipient object where Message.BoxType = egwDraft	AddressBookEntry, GroupMember objects
Members	GroupMembers collection			R/O (collection). Individual GroupMember objects in this collection are R/W. GroupMember objects can be added with the GroupMembers.Add() method or removed with GroupMember.Delete().
ObjType	Enum	R/O		R/W
Parent	Object, specific object varies.	R/O	R/O	R/O
Resolved	Enum		R/W - can be reset to egwNotResolved, but not to other values	
TargetType	Enum		R/W	R/O - GroupMember only
Methods		(none)	Delete() Resolve()	Delete() Refresh()

Address Books

The Address Books object is a collection of Address Books. The simplified hierarchy of Address Books in GroupWise is:

```

Account object
  AddressBooks collection
    AddressBook object
      AddressBookEntries collection
        AddressBookEntry object
          GroupMembers collection
            GroupMember object (if AddressBookEntry.ObjType = egwGroup)

```

The Account object is returned by one of the 3 methods of the Application object covered in Chapter 1: *Introducing the Object API*.

The AddressBooks collection consists of several different address books. The primary AddressBook is the System address book. Some or all of the following address books are available in the AddressBooks collection:

Address Book	Type	Subtype
System	egwNovellSystem	egwOther
Personal	egwNovellPersonal	egwOther
Frequent contacts	egwNovellPersonal	egwFrequentContact
LDAP	egwUnknownBook	egwOther
MAPI	egwUnknownBook	egwOther
Shared (Personal)	egwNovellPersonal	egwOther

The AddressBooks collection has the usual collection properties:

Property	Data Type	Description
Application	Application object (R/O)	The Application object
Count	Long (R/O)	Number of objects in the collection.
_NewEnum	Object (R/O)	Enumeration object implementing IEnum-VARIANT for Windows only.
Parent	Account object (R/O)	The Account that owns this collection.

It also supports the following two methods:

```
AddressBooks.Add(Name as String)
```

This method allows the user to create a new “personal” address book. It returns a new AddressBook object. Remember to put an appropriate variable on the left side of the statement to catch the return Address Book object (using your language’s method of doing so). Add does not support Before or After parameters in VB.

```
AddressBooks.Item(Index as {Long or String})
```

This method returns an existing AddressBook object. If Index is a Long, Item() returns the AddressBook object at the given index in the collection. If Index is a String, Item() returns the AddressBook with the same Name as Index.

You may also obtain an `AddressBook` via the `Account.SystemAddressBook`, the `Account.DefaultAddressBook`, or `Account.FrequentContacts` properties.

Note: `AddressBooks` does not have a `Remove` method. Use the `Delete()` method of the individual `AddressBook` objects instead.

Sharing an AddressBook

With GroupWise 5.5 and later it is possible to accept shared address books using the Object API. To do this, you need to use the new `SharedNotification` object. The `SharedNotification` object is a new subtype of the `Message` class. It is the message subtype that is received when some other user wishes to share an address book with the recipient.

Since the `SharedNotification` object is a subtype of the `Message` object, it includes the same properties and methods as the `Message` object. In addition, it adds three additional properties: `Description`, `Name`, and `NotificationType`. It also has two methods called “Accept” that enable a user to accept either shared address books or folders.

This particularly type of message is NOT a message that can be created from scratch using the Object API. Its properties are “read only”. It needs to be created through the GroupWise Client. However, it has a “`Message.Classname`” that can be recognized by the recipient of the shared notification, and then accepted through the Object API. This `Classname` is “`GW.MESSAGE.MAIL.NEW.SHARED.PAB.NOTIFY`”

To accept a shared address book with this new object, you have two options:

1. Use the `Accept` method of the `SharedNotification` object.
`ShareNotification.Accept (String SharedAddressBookName, [String Description])`
2. Use the new `AddEx` method of the `AddressBooks` object.
`AddressBooks.AddEx(SharedABNotify as SharedNotification, [String sName])`

The `AddEx` method above actually calls the `Accept` method in the internal GroupWise code.

Here is an example in Visual Basic of how a user would accept a Shared Address Book using the Object API. It demonstrates both options outlined above.

```
Dim Acc As GroupWareTypeLibrary.Account2
Dim MyFolder As GroupWareTypeLibrary.Folder2
Dim MyMessages As GroupWareTypeLibrary.Messages
Dim MyMessage As GroupWareTypeLibrary.Message2
Dim gwAddressBooks As GroupWareTypeLibrary.AddressBooks2
Dim gwShareMessage As GroupWareTypeLibrary.SharedNotification
```

```

Dim gwNewAB As GroupWareTypeLibrary.AddressBook2

Dim i As Integer
Dim MsgName As String
Dim MsgDesc As String
Dim MsgType As Integer
Dim SomeInt As Integer
Dim MsgSubject As String

Dim GWApp As New Application2

Set Acc = GWApp.Login
Set MyFolder = Acc.MailBox
Set MyMessages = MyFolder.Messages

Set gwAddressBooks = Acc.AddressBooks

For i = 1 To MyMessages.Count
    Set MyMessage = MyMessages.Item(i)

'Find a SharedNotification message.
    If MyMessage.ClassName = "GW.MESSAGE.MAIL.NGW.SHARED.PAB.NOTIFY" Then
'Demonstrate how to get the properties of the SharedNotification message
        Set gwShareMessage = MyMessages.Item(i)
        MsgName = gwShareMessage.Name
        MsgDesc = gwShareMessage.Description
'The MsgType should be equal to egwSharedAddressBook
        MsgType = gwShareMessage.NotificationType

'Either of the two methods below will accept a shared address book. Uncomment
'one of them. Here, some additional code is added just to check on the subject
'of the shared address book. It may not be needed.

        '
        '    MsgSubject = gwShareMessage.Subject.PlainText
        '    If (MsgSubject = "Sharing of Address Book 'mynewbook'.") Then
        '        Set gwNewAB = gwAddressBooks.AddEx(gwShareMessage)
        '    End If

        '
        '    MsgSubject = gwShareMessage.Subject.PlainText
        '    If (MsgSubject = "Sharing of Address Book 'mynewbook'.") Then
        '        gwShareMessage.Accept ("mynewbook")
        '    End If

        End If
    Next

```

Summary

In this chapter you learned about different Address objects; Address, Recipient, AddressBookEntry, and GroupMember. You learned about methods and properties of the AddressBook object that enable you to add, modify and delete new entries. In addition, you learned about Shared Address Books. In the next chapter you will learn about Field and Field related objects.

For more information on Address and AddressBook objects, please check the GroupWise Object API documentation <http://developer.novell.com/ndk/doc.htm>, sample code <http://developer.novell.com/ndk/sample.htm> or visit the Developer Support Forum area at <http://developer-forums.novell.com/category/index.tpt>.

Copyright © 2002 by Novell, Inc. All rights reserved.
No part of this document may be reproduced or transmitted
in any form or by any means, electronic or mechanical,
including photocopying and recording, for any purpose
without the express written permission of Novell.

All product names mentioned are trademarks of
their respective companies or distributors.

Understanding Field and Field Related Objects

Chapter 8

Section 1: GroupWise Object API

Field objects, contained in Fields objects, in GroupWise represent a user-defined field in an AddressBookEntry, Document, or Message object. In an AddressBookEntry or Document, it can also represent a predefined field.

FieldDefinitions objects contain the FieldDefinition objects which define the fields for an Account, AddressBook or DocumentLibrary. The root account and all AddressBooks have an independent collection of FieldDefinitions. The root account also includes user-defined fields that can appear in a message. An account collection includes user-defined and predefined fields that can appear in its entries.

Finally, LookupTableEntry objects, referenced in LookupTableEntries, are used to provide restricted values a field.

The following topics are discussed in this chapter.

Contents:

- Pre-defined and Custom Field Objects
- Accessing Field Information
- Custom Fields and the GroupWise Client
- Deleting Fields and FieldDefinitions
- Custom Fields in the Address Book
- Fields and Document Libraries
- Lookup Tables
- Related Properties
- Summary

Pre-defined and Custom Field Objects

Although the information in most pre-defined fields can be accessed through a named data member that is associated with a specific GW object, GroupWise also allows you to directly access the information using a Field object. In addition, GroupWise allows you to create fields of your own, referred to as “custom fields”. A custom field is also an object of type “Field”. GroupWise stores its Field objects in a collector object called “Fields”.

Only the following GroupWise objects have a “Fields” property:

- Objects derived from the base Message class (including mail messages, appointments, tasks, notes, phone messages, document references, and custom message types defined via the C3PO API).
- Folder object
- Document object
- AddressBookEntry object

Each time GroupWise adds a custom field to an object, it needs to know what kind of field it is, and what its name should be. Therefore, before you create a custom field, you must first create a “template” for the field called a FieldDefinition.

Two GroupWise objects can contain custom fields based on the same FieldDefinition, but there are properties of the custom fields that are unique to the object that owns the field, such as the value of the field. A Field object contains a property that indicates which FieldDefinition the Field object is based on.

A FieldDefinition describes many aspects of the new field, including what it should be called, what kind of data it can contain, how much data it can contain, and whether or not it should be visible in a user interface. FieldDefinition objects are contained in FieldDefinitions collections.

Only the following GroupWise objects have a “FieldDefinitions” property:

- Account Object
- DocumentLibrary Object
- AddressBook Object

Before you can create a custom field for an object, you must first define the Field in the FieldDefinitions property of the object’s parent. To do this, use the FieldDefinitions.Add() method. The one exception to this is the DocumentLibrary object. DocumentLibrary objects have FieldDefinitions properties, but to create field definitions for a DocumentLibrary, you must use the GroupWise Admin API.

To create a custom field for a GroupWise object, you call “Add()” of that object’s “Fields” property. This method takes three parameters: the name of the field, a constant defining the type of the field, and the value of the field. The name of the field must match the name of a FieldDefinition object in the FieldDefinitions property of the GroupWise object’s parent. For a Message or Folder object, the parent is the Account object to which it belongs. For a Document object, the parent is the DocumentLibrary object to which it belongs. And, for an AddressBookEntry object, the parent is the AddressBook object to which it belongs. The type constant passed to the Fields.Add() method must match the defined type of the FieldDefinition referenced in the name parameter otherwise an exception is generated. The value is what you intend to store in the Field.

To add a field to the Fields collection object referred to by the following properties...	...We must first insure a FieldDefinition exists in these FieldDefinitions collections.
Message.Fields	RootAccount.FieldDefinitions
Folder.Fields	RootAccount.FieldDefinitions
AddressBookEntry.Fields (AddressBookEntries that are groups, ObjType = egwGroup, do not support a Fields collection)	AddressBook.FieldDefinitions
Document.Fields	DocumentLibrary.FieldDefinitions

The key FieldDefinition properties are:

Property	Data Type	Description
FieldID	Long (R/O)	Not yet implemented. Field identifier.
FieldType	Enum (R/O)	Type of field defined by the FieldDefinition object.
HasLookupTable	Boolean (R/O)	TRUE if the FieldDefinition has a lookup table.
Hidden	Boolean (R/O)	TRUE if the field will not show up in any standard GroupWise use interface.
MaximumLength	Long (R/O)	Maximum string or BLOB length of the field.
MinimumValue, MaximumValue	Long (R/O)	Minimum or maximum value of a numeric field.
Name	String (R/O)	Name of the field being defined. Value is case-sensitive.
ReadOnly	Boolean (R/O)	TRUE if the field value cannot be changed.
RelatedFieldDefinition	FieldDefinition object (R/O)	A FieldDefinition object which has constraint values for this field. Constraint values can be stored in a single FieldDefinition object, simplifying maintenance.
Required	Boolean (R/O)	TRUE if a value is required for this field.
StringCase	Enum (R/O)	egwUpperCase, egwLowerCase, egwMixedCase.

Adding a `FieldDefinition` to a collection is simple.

```
FieldDefinitions.Add(Name as String,FieldType as Enumerated Integer)
```

This method uses the first parameter to name the new `Field`. Because this value is case sensitive, you may use the same name, so long as the type differs; however, this is not a good idea in practice. The second parameter defines the value that can be held by the `FieldDefinition`.

The following `FieldTypes` are defined:

FieldType	Enum value	Description
EgwString	1	String up to 64K.
EgwNumeric	2	A Long, 32 bit, integer. Floating point numbers are not currently supported, and any decimal values will be truncated. Pass a Single or Double value by converting it to a String.
EgwDate	3	This has a <code>VarType = 7</code> , or date variant type. Use appropriate representation according to the operating system locality.
EgwBinary	4	BLOB up to 64K.
EgwReserved	5	Reserved for future use.

Example in VB:

```
'gwAccount is a valid Account object  
  
Set gwFieldDefs = gwAccount.RootAccount.FieldDefinitions  
gwFieldDefs.Add "MyString", egwString  
gwFieldDefs.Add "MyNumeric", egwNumeric  
gwFieldDefs.Add "MyDate", egwDate  
gwFieldDefs.Add "MyBinary", egwBinary
```

The above code works great the first time, but crashes the second time. This is because we are attempting to create a new member of a collection where one already exists. So to be nice to our programs and users, the code should actually be:

Example in VB:

```
'gwAccount is a valid Account object  
  
If gwFieldDefs.Item("MyString", egwString) Is Nothing Then  
    gwFieldDefs.Add "MyString", egwString  
End If
```

Actually, we can create a useful Sub to remember to do this for us. The one that follows assumes the `gwRootAccount` is defined at the module or global level.

Example in VB:

```
'gwRootAccount is a valid Account object

Public Sub AddFieldDefToRootAccount(sFieldName as String, eFieldType as Long)
If Not gwRootAccount Is Nothing Then
    With gwRootAccount.FieldDefinitions
        If .Item(sFieldName, eFieldType) Is Nothing Then
            .Add(sFieldName, eFieldType)
        End If
    End With
End If
End Sub
```

In the following code snippet, we define a custom field for Message and Folder objects. We then create the Field for a specific folder and all of the messages in that Folder.

Here is the sample. Note the absence of error handling. This is for simplicity.

Example in VB:

```
'gwAccount is a valid Account object

Private Sub CreateProjectIDFieldDefinition()
'the following line creates a field definition called ProjectID with type string
    gwAccount.FieldDefinitions.Add "ProjectID", egwString
End Sub

Private Sub AddProjectIDField()
    Set gwCabinetFolder = gwAccount.Cabinet
    Set gwProjectFolder = gwCabinetFolder.Folders.ItemByName("Project 123")
    gwProjectFolder.Fields.Add "ProjectID", egwString, "123"

'the following command allows us to get the count once, rather than placing it
'in the "for" loop, and letting the overhead of COM kill us over and over again...
    jCounter = gwProjectFolder.Messages.Count

    For iCounter = 1 To jCounter
        Set gwMessage = gwProjectFolder.Messages.Item(iCounter)
        gwMessage.Fields.Add "ProjectID", egwString, "123"
    Next iCounter
End Sub
```

Example in Delphi:

```
//gwAccount is a valid Account object

procedure CreateProjectIDFieldDefinition;
begin

// the following line creates a field definition called ProjectID with type
  string gwAccount.FieldDefinitions.Add( 'ProjectID', egwString );
end;

procedure AddProjectIDField;
var
  gwCabinetFolder : Variant;
  gwProjectFolder : Variant;
  gwMessage : Variant;
  iCounter : Integer;
  jCounter : Integer;

begin

gwCabinetFolder := gwAccount.Cabinet;
  gwProjectFolder := gwCabinetFolder.Folders.ItemByName( 'Project 123' );
gwProjectFolder.Fields.Add( 'ProjectID', egwString, '123' );

  // the following command allows us to get the count once, rather than
  placing it
  // in the "for" loop, and letting the overhead of COM kill us over and
  over again...
  jCounter := gwProjectFolder.Messages.Count;

  for iCounter := 1 to jCounter do begin

    gwMessage := gwProjectFolder.Messages.Item(iCounter);
    gwMessage.Fields.Add( 'ProjectID', egwString, '123' );

  end; // for

  gwCabinetFolder := unassigned;
  gwProjectFolder := unassigned;
  gwMessage := unassigned;

end;
```

At this point, we are able to add whichever of these fields we want to a new message.

Note: The field values are read/write as long as the message is read/write. This is the case only for BoxType of egwDraft and egwPersonal messages.

Values of fields must be initialized, therefore the Add () method requires all three parameters, the third of which is the initial value.

GroupWise currently allows the Add () method to be used multiple times with the same FieldName and FieldType without raising an error; it just overwrites the previous entry.

Example in VB:

```
'gwAccount is a valid Account object

Dim gwNewMessage As GroupwareTypeLibrary.Mail3

Set gwNewMessage = gwAccount.WorkFolder.Messages.Add _
    ("GW.MESSAGE.MAIL", egwDraft)

With gwNewMessage
    .Subject = "Fields Test Message"
    .BodyText.PlainText = "Hello, here is a message with embedded fields."
    .Recipients.AddByDisplayName("Jane Doe", egwTo)
End With

Dim gwField As GroupwareTypeLibrary.Field2
Set gwField = gwNewMessage.Fields.Add("MyString", egwString, "Yes")
` Oh, I changed my mind
gwField = "No"
` I really can't decide
gwField.Value = "Maybe"
` I can also add a field without returning an object
gwNewMessage.Fields.Add "MyDate", egwDate, #2/15/2000#
` note the lack of parenthesis
` now it's time to send the message
` (Lets hope you changed Jane Doe to someone who exists).
gwNewMessage.Send
```

The `Field` object has two methods.

```
Field.Delete ()
```

This method deletes a field from an `AddressBookEntry`, `Document`, or `Message`.

```
Field.Validate (Value as Variant)
```

This method returns a Boolean, that is `TRUE` if 'Value' is a valid value for this field as defined in the `FieldDefinition`, or `FieldDefinition.RelatedFieldDefinition` objects.

Accessing Field Information

Accessing the information in custom fields is fairly simple once you understand how custom fields are stored for a GroupWise object. You access a custom field for an object through that object's `Fields` property. The `Fields` property has two `Item()` methods.

The first form of `Fields.Item()` takes two parameters: a string that represents the name of a field, and a field type constant. If you know the name and type of the field you want to access, this is the method you want to use. Below is an example.

Example in VB:

```
Set gwProjectIDField = gwMessage.Fields.Item("ProjectID", egwString)
```

Example in Delphi:

```
gwProjectIDField := gwMessage.Fields.Item('ProjectID', egwString);
```

If you want to forage through all of the custom fields for an object, either as an iteration exercise or as a discovery exercise, then you would use the second form of the `Fields.Item()` method. This form takes an integer as a parameter. The integer acts as an index into the list of custom fields. `Fields.Item(1)` would return the first custom field in the list, `Fields.Item(2)` would return the second. If the integer is less than 1 or greater than `Fields.Count`, an exception is generated. Below is an example.

Example in VB:

```
For j = 1 to gwMessage.Fields.Count
    Set gwProjectIDField = gwMessage.Fields.Item(j)
    ' do something with the field
Next j
```

Example in Delphi:

```
for j := 1 to gwMessage.Fields.Count do begin
gwProjectIDField := gwMessage.Fields.Item(j);
    // do something with the field
end;
```

If the number, names and types of custom fields that exist for a GroupWise object are not known, using this form of `Fields.Item()` is useful. Once you have a custom field object, you can examine its definition. The `Field` object has a `FieldDefinition` property that corresponds with the `FieldDefinition` object in the parent's `FieldDefinitions` collection that the custom field is based on. From this, you can determine the field's type and name, value restrictions, etc. Once you know the field's type, you are prepared to deal with its value. For instance, the following code snippet shows how to determine a field's type before using its value. In this example, `ebFieldName` is an edit box on a form, and `lblFieldType` is a label object.

Example in VB:

```
'gwMessage is a valid Message object

Set gwProjectIDField = gwMessage.Fields.Item(1)
' note that Field.Name = Field.FieldDefinition.Name
```

Each column represents a property of message items and folder objects that are displayed in the pane. By right-clicking on the column and selecting the “More Columns” menu item from the pop-up menu, you will be presented with a list of all pre-defined fields as well as custom fields defined in the FieldDefinitions property for the mailbox. Any defined field, whether pre-defined or custom, can be displayed in a column in the GroupWise client. What’s more, since custom fields must be one of four types (string, numeric, date, binary), items can be sorted in ascending or descending order according to the contents of the custom field.

The only exception here is Binary custom fields. If you create a custom field of type egwBinary, that custom field will not show up in the More Columns display in the GroupWise client. Binary custom fields cannot be displayed by the GroupWise client because their data cannot be interpreted as anything by the GroupWise client but raw binary data.

Note that you can create custom fields for folder objects. If you have multiple folders under the Cabinet folder, and you click on the cabinet folder, the child folders it contains will be displayed in the right-hand pane. If any of those folders has a custom field that is being displayed in a column, the value for that field will be displayed for that folder, just as it is for any other object displayed in the GroupWise client.

Finally note that when a FieldDefinition is added to any FieldDefinitions object, whether for an Account object, a DocumentLibrary object (through the Admin API), or an AddressBook object, that FieldDefinition object is accessible by any account in the same post office as the account that created it. In fact, all field definitions are stored together at the post office level, regardless of whether they were created as part of the Account.FieldDefinitions collection, the DocumentLibrary.FieldDefinitions collection, or the AddressBook.FieldDefinitions collection. The Object API documentation calls this a “subterfuge”. The fact that only a subset of the FieldDefinition objects can be accessed by the Account.FieldDefinitions object creates the illusion that those FieldDefinition objects are somehow stored separately from DocumentLibrary.FieldDefinition objects and AddressBook.FieldDefinition objects, when in fact they are all stored together at the post office level.

Deleting Fields and FieldDefinitions

Suppose you have added a custom field to a GroupWise object, and you now want to remove it. In order to do this, you must retrieve the Field object itself, and call its own Delete() method, as demonstrated below.

Example in VB:

```
Set gwField = gwMessage.Fields.Item( "ProjectID", egwString )  
gwField.Delete
```

Example in Delphi:

```
gwField := gwMessage.Fields.Item( 'ProjectID', egwString );  
gwField.Delete;
```

Likewise, if you want to remove a custom FieldDefinition, retrieve the FieldDefinition object and call its Delete() method, as shown below.

Example in VB:

```
Set gwFieldDefinition = gwAccount.FieldDefinitions.Item("ProjectID",egwString )  
gwFieldDefinition.Delete;
```

Example in Delphi:

```
gwFieldDefinition := gwAccount.FieldDefinitions.Item('ProjectID',egwString );  
gwFieldDefinition.Delete;
```

If you delete a FieldDefinition without first deleting the existing custom fields upon which it is based, GroupWise appears to have removed the custom fields. For example, suppose you have a FieldDefinition object called “ProjectID”. Suppose further that a message in a GroupWise mailbox has a custom field based on the ProjectID FieldDefinition object. If you execute FieldDefinition.Delete() on the ProjectID FieldDefinition object, and then examine the Fields collection of the GroupWise object, it will appear that the Field object that was based on the ProjectID FieldDefinition has been deleted, even though you never executed the Field.Delete() method.

Custom Fields in the Address Book

In order to fully understand how custom fields work for AddressBookEntry items, you need to have an understanding of AddressBook, AddressBookEntries, and AddressBookEntry items. For information on SharedNotifications, see Chapter 7: *Understanding Address and AddressBook Objects*.

GroupWise supports two types of address books: System and Personal. There is always only one System address book. The AddressBookEntry objects that exist in the System address book are read-only. You cannot add custom field definitions to the System address book AddressBook.FieldDefinitions collection through the Object API. Instead, you must use the Admin API to invoke AddressBook.FieldDefinitions methods. With the Object API, you can only access custom fields and field definitions for the System address book as read-only objects.

Personal AddressBook objects can be modified in any way using the Object API. You should note that there are two types of Personal address books: GroupWise Personal Address Books, and External MAPI Personal Address Books.

The GWORB manages address book items using MAPI. The GroupWise System Address Book and GroupWise Personal Address Books are all installed as MAPI Service Providers in the Windows Messaging subsystem on a Win32 machine. The GroupWise client engine accesses and manages GroupWise address objects via MAPI. Because of this, you can install other MAPI Address Book Service Providers and GroupWise will access and manage them just like its own address books. Note, however, that GroupWise can tell the difference between its GW system address book, native GW personal address books, and external MAPI address books.

MAPI is fairly flexible when it comes to defining properties of address book objects. It does provide a mechanism whereby MAPI Address Book Service Providers can define and maintain non-standard fields for address book objects. Each service provider is free to define these fields and maintain them as they see fit.

As an example, the GroupWise Personal Address Book Provider allows you to create multiple “additional fields”. If you look at the presentation of object data on a user in a GroupWise personal address book, you will see a button labeled “Advanced”.

The screenshot shows a dialog box titled "Information for Jonathan Jett". It contains the following fields and controls:

- First name: Jonathan
- Last name: Jett
- Display name: Jonathan Jett
- Organization: Tangible Solutions
- E-mail address: JJett@TangibleSolutions
- Department: R&D
- E-mail type: (dropdown menu)
- Title: Director of R&D
- Address: P.O. Box 907768
- Mailstop: (empty field)
- Greeting: (empty field)
- City: Orem
- Phone #s: Office (801) 221-7606
- State: UT
- ZIP code: 84097
- Country: USA
- Comments: (empty text area)

Buttons: OK, Cancel, New, Info, Format, Advanced...

Figure 2: The data on a user in a GroupWise personal address book.

Clicking this button presents an interface that lets you define new fields for address book items, as well as access and modify values for any custom fields for this specific address book item. Although this capability is supported by MAPI, it is exposed in this manner by the GroupWise Personal Address Book MAPI Service Provider. Not all MAPI address books will provide you with this capability.

As an example, let's look at the Microsoft Personal Address Book MAPI Service Provider. When this service provider is installed and configured, a new address book is added to the collection. It is labeled "Personal Address Book", as shown in the graphic below:

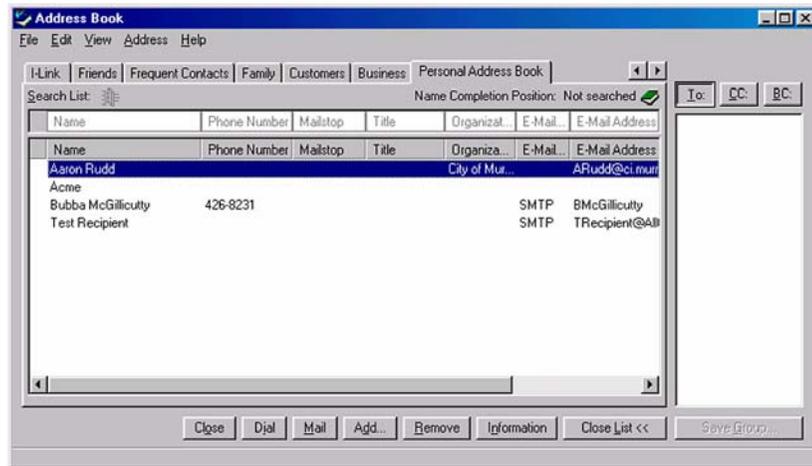


Figure 3: *The Microsoft Personal Address Book MAPI Service Provider.*

From the outset, there isn't much difference between this columnar view of data in the Microsoft Personal Address Book and the native GroupWise Personal Address Books. But when you look at the details of an address book item in the Microsoft PAB, it looks like the screen in Figure 4.

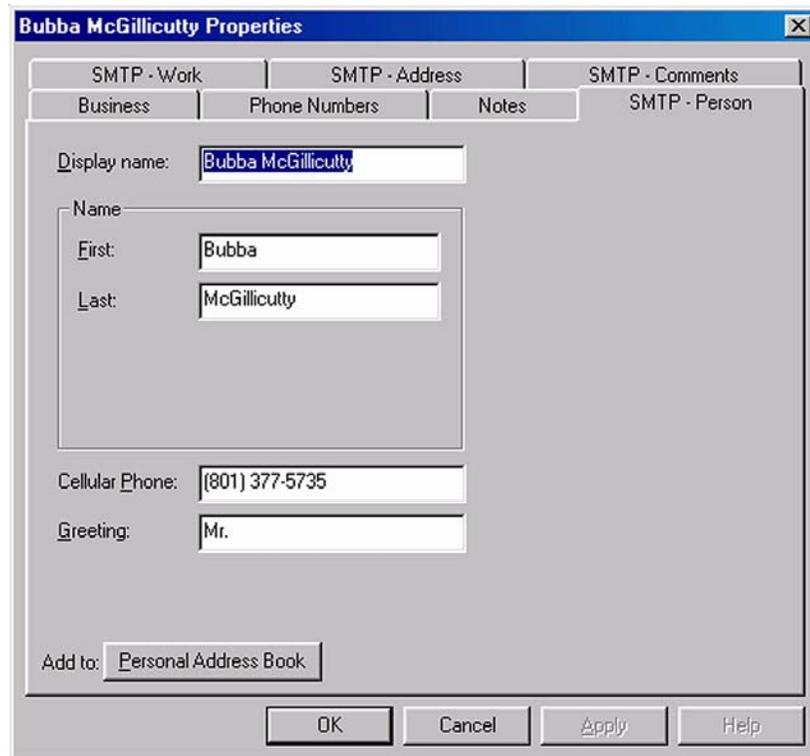


Figure 4: *The Microsoft PAB.*

There is a big difference! In addition to different tabs, this view does not provide the capabilities described above, i.e. the creation, access, and manipulation of custom fields. The Object API does not distinguish between GroupWise address books and non-GW MAPI address books when it comes to dealing with custom fields. You can access the Fields collection on an address book item that belongs to a non-GW address book just as you can for items belonging to a native GW address book. But note also that the list of field definitions and fields that exist for a non-GW address book and its items will be different than those for native GW address books.

For example, if you right-click on the column manager for a native GroupWise personal address book, and select the “More Columns” menu item, you see something similar to Figure 5.

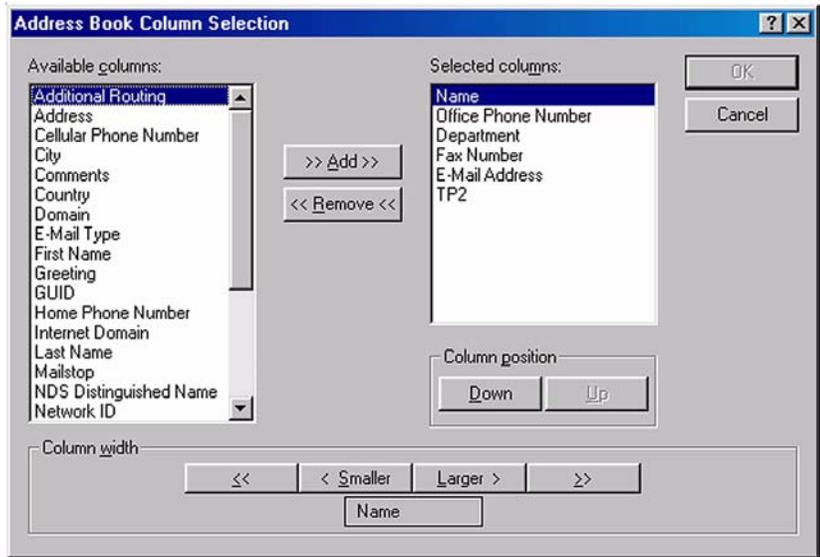


Figure 5: Address Book Column selection.

Note the list of “available” and “selected” columns. Now, if you right-click on the column manager for the Microsoft Personal Address Book, and select “More Columns”, you the screen shown in Figure 6.

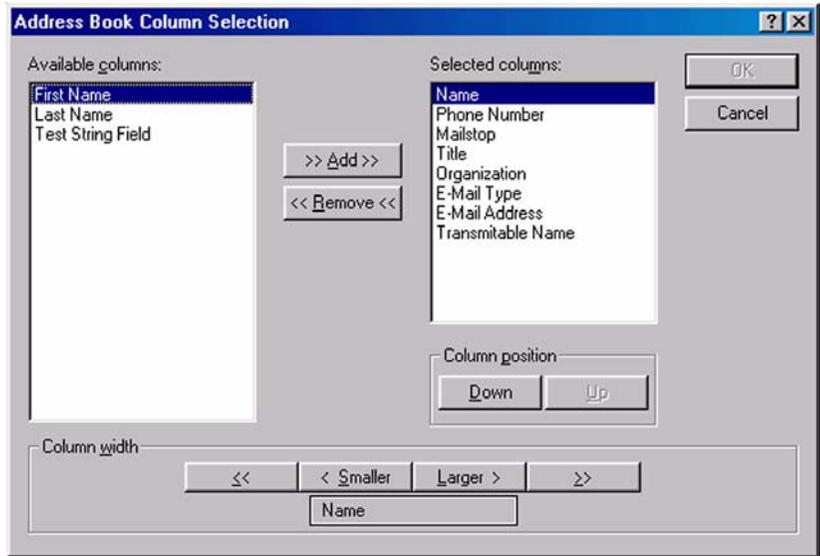


Figure 6: Address Book Column selection.

It's quite a different list! The list of fields in this dialog corresponds with the FieldDefinition objects you would see in the FieldDefinitions collection for the address book. The GroupWise personal address book has quite an impressive list of fields, and quite unique as well. One way, though certainly not the only way, to determine if a specific address book item is a GroupWise address book or not is to examine its FieldDefinitions container for tell-tale fields such as "NDS Distinguished Name" or "Domain".

While the Microsoft PAB provider doesn't give you an interface to create custom fields as the GW PAB provider does, you are still free to add FieldDefinition objects to the MS PAB FieldDefinitions collection using the Object API, just as you can with the GW address books. Remember that the API does not distinguish between GW address books and non-GW address books when you call methods dealing with custom fields for address books.

While these fields cannot be accessed or modified in the "Information" screen provided by the MS PAB provider, you can display the field in a column, as demonstrated above. However, note that in the authors' testing, calling the AddressBook.FieldDefinitions.Add() method for the MS PAB generated a protection fault and forced a reboot of the Windows OS. You might consider the creation and manipulation of custom fields for non-GW address books to be something of an experiment. We aren't sure what will happen, or if it is supposed to work! But you might like trying it!

The GroupWise address books can contain four different types of address book items:

- Users
- Resources
- Organizations
- Groups

Users, Resources, and Organizations can all have custom fields. Only Groups (also known as "Distribution Lists" can NOT have custom fields. Attempting to access or otherwise manipulate FieldDefinitions or Fields objects in conjunction with Group items will generate an exception.

Fields and Document Libraries

Document Objects versus Document References

Each Document Library in GroupWise is a special database that contains document objects. A user's GroupWise mailbox can contain Document References that point to specific document objects in any of the Document Libraries in the system. It is necessary to distinguish between a document object and a Document Reference.

As discussed in Chapter 6: *Understanding Document and Document Related Objects*, a DocumentReference is a subtype of the GroupWise Message object, just like an Appointment or Task object. As such, a DocumentReference is an object in its own right, and resides along with other descendants of the base Message class in the GroupWise mailbox. As a message object, a DocumentReference can contain custom fields that are defined by the FieldDefinitions property of the Account object, just as GroupWise folders, mail messages, and appointments can do.

Each DocumentReference object also contains fields that indicate which document object it references, or points to.

The document object is also a database object in its own right, and is contained in the Document Library database(s). Each document object can also have custom fields that contain useful information. But just as Message objects in a GroupWise mailbox may only have custom fields that are based on FieldDefinition objects in the Account.FieldDefinitions collection, document objects may only have custom fields that are based on FieldDefinition objects in the DocumentLibrary.FieldDefinitions collection.

The FieldDefinitions property of any DocumentLibrary object only contains custom field definitions. It does not contain field definitions that are pre-defined, such as "Author" or "Document Type". These fields already exist for every document object in each library. This contrasts with Account.FieldDefinitions which contains all FieldDefinition objects for the entire post office and AddressBook.FieldDefinitions which contains FieldDefinition objects for all pre-defined and custom fields for AddressBookEntry objects.

While you can access custom field information for DocumentLibrary and Document objects using the Object API, you cannot create custom FieldDefinition or Field objects using the Object API. You must use the GroupWise Administrator plug-in for NWADMN32.EXE, or use the GroupWise Admin API to create custom FieldDefinition and Field objects for DocumentLibrary.FieldDefinitions and Document.Fields collections.

To explain further how custom fields work for DocumentLibrary and Document objects, we use the following example.

Suppose you have a DocumentLibrary named “Legal”. It is the library where all legal documents are kept. You would like to have a field that contains a case number identifying which case a particular document is related to. Using the GroupWise Admin API, you can create a new field called “Case ID”. To do this with the GW Administrator, the administrator would go to the “Document Properties Maintenance” screen, as shown in Figure 7.

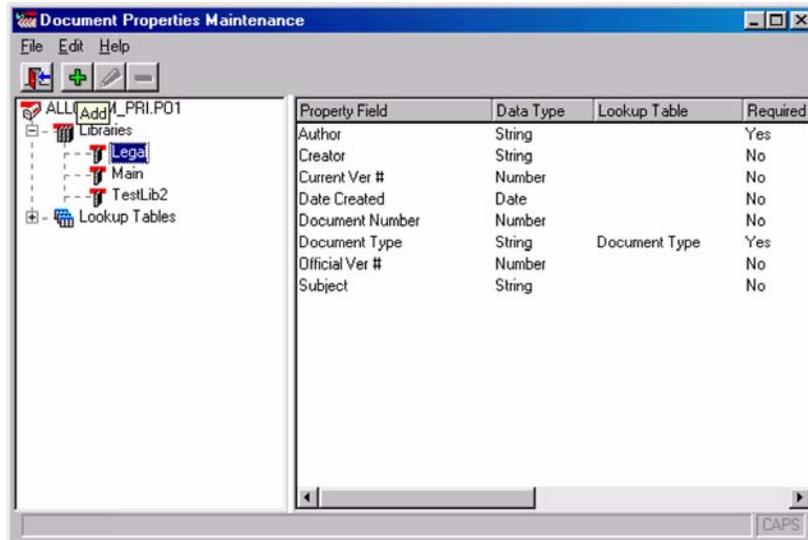


Figure 7: Document Properties Maintenance.

Click on the “Add” button (the big green plus sign on the tool bar). This will present the screen shown in Figure 8.

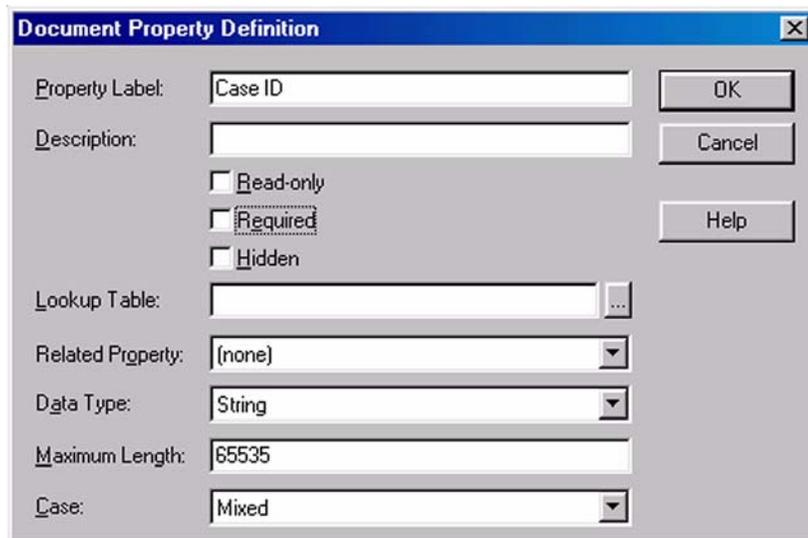


Figure 8: Document Property Definition.

Filling in the Property Label with “Case ID” and accepting the defaults, this creates a new string field called “Case ID” for documents in the Legal library. This is essentially adding a new FieldDefinition to the DocumentLibrary.FieldDefinitions collection of the Legal library. The name of the new FieldDefinition object would be “Case ID”.

After doing this, the user would be able to populate this new field for new documents in the Legal library, as shown in Figure 9.

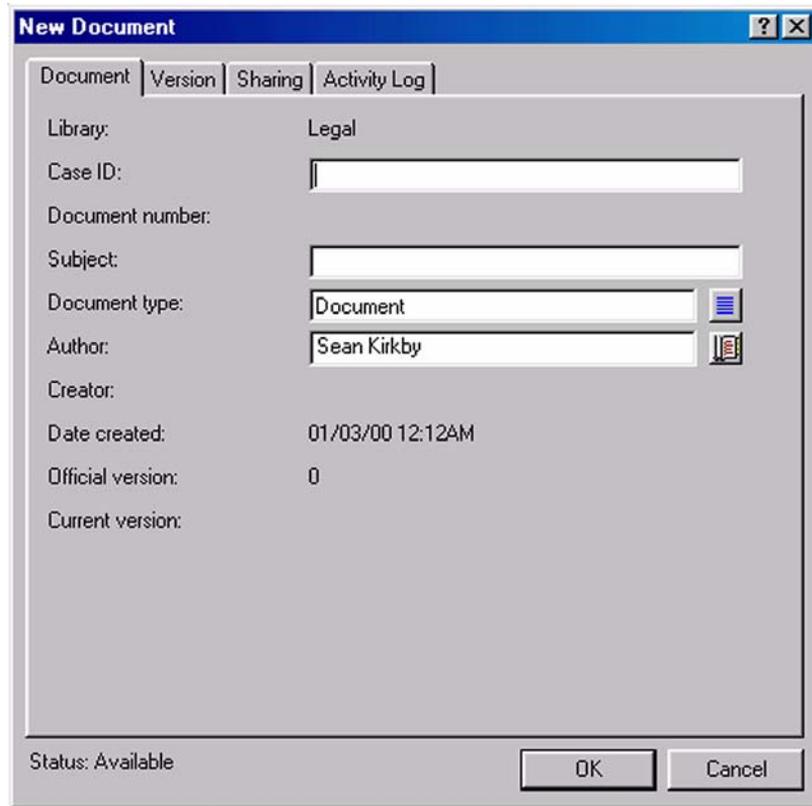


Figure 9: *New Document.*

You could also write an application that uses the Object API to set and read the case ID for a particular document.

Example in VB:

```
`gwDocument is a valid Document object  
  
Set gwCaseIDField = gwDocument.Fields.Item("Case ID", 1)  
gwCaseIDField.Value = "990304-3432"
```

Example in Delphi:

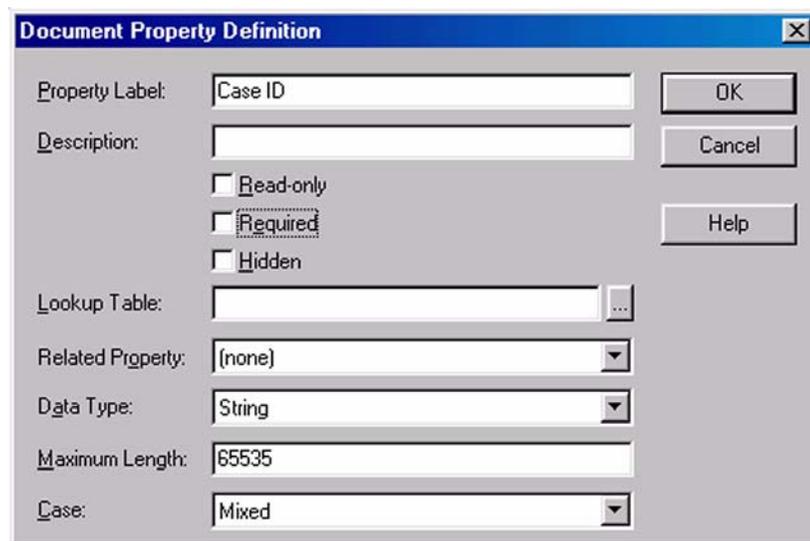
```
//gwDocument is a valid Document object  
  
gwCaseIDField := gwDocument.Fields.Item('Case ID', egwString);  
gwCaseIDField.Value := '990304-3432';
```

Lookup Tables

As a Document Management System administrator, you can imagine the danger of requiring end users to populate the Case ID field manually for each document that is created. If it were possible, it would be desirable to force the user to select from a list of case IDs when selecting a case ID for a particular document. That way, they couldn't accidentally type in a case ID that was not valid.

GroupWise allows for this using a concept called a “lookup table”. The administrator can create a “mini-database” called a lookup table and populate it with pertinent values. When the administrator creates a custom field using the GroupWise Administrator, he or she can also associate that field with the lookup table. Doing so changes the display of the field for the end-user from a free-form field, like an edit box, to a “combo-box” field, also known as a “drop-down list box”. The administrator can populate the lookup table using the GroupWise administrator.

Figure 10 is a view of how a lookup table is created.



The screenshot shows a dialog box titled "Document Property Definition". It contains the following fields and controls:

- Property Label:** Text box containing "Case ID".
- Description:** Empty text box.
- Read-only:** Unchecked checkbox.
- Required:** Unchecked checkbox.
- Hidden:** Unchecked checkbox.
- Lookup Table:** Empty text box with a browse button (...).
- Related Property:** Dropdown menu showing "(none)".
- Data Type:** Dropdown menu showing "String".
- Maximum Length:** Text box containing "65535".
- Case:** Dropdown menu showing "Mixed".
- Buttons:** OK, Cancel, and Help buttons on the right side.

Figure 10: How a lookup table is created.

Select the “Lookup Tables” item in the left-hand pane, and click the “add” button on the tool bar. This brings up the dialog box shown in Figure 11.

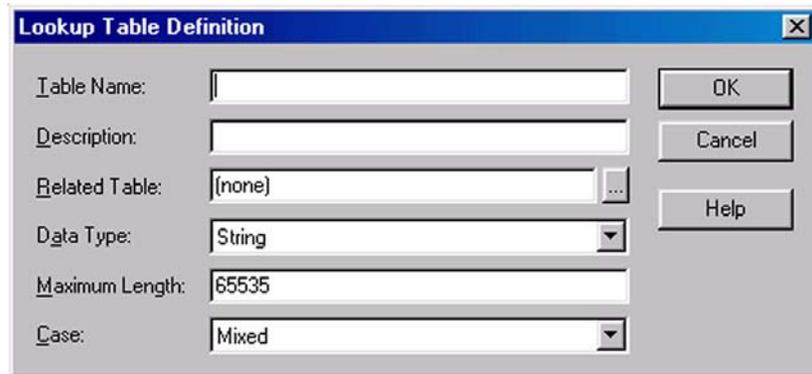


Figure 11: *Lookup table definition.*

Here, the administrator can give the table a name (like “Case ID Values”), and determine what type of data the table can hold (string or numeric). In addition, the maximum length and allowed case for each value in the table can be set for strings, and the minimum and maximum values for numbers.

Once the table has been created, the administrator can modify the table by adding values. This is done by selecting the table and clicking the “add” button to bring up the dialog box in Figure 12.



Figure 12: *Lookup entry.*

The administrator can create values and add them to the table.

Once this is done, the administrator can create a custom field and link it with the table. When the new field is being created, the administrator selects the “Lookup Table” selection icon, and the following prompt is presented (see Figure 13).



Figure 13: *Lookup table list.*

In this example, a number of lookup tables have been created. By default, there is only one pre-defined lookup table. It's called "Document Types", and it is already linked with a pre-defined field called "Document Type". When users save documents, they must select from the list of values in the Document Types lookup table. The administrator can modify the list by adding custom document types and removing pre-defined types.

When a field is linked to a lookup table, the field's data type and other parameters cannot be modified. The field must take on the parameters of the lookup table. For instance, if you link a field to a lookup table that has string values in it, the field must be a string field. Furthermore, if the values in the lookup table can only be ten characters long and must be upper case, the field must take on these limitations as well.

When a field has been linked with a lookup table, the field is not displayed as a free-form field as demonstrated above. Rather, it is displayed as a box with a selector, as shown in Figure 14.

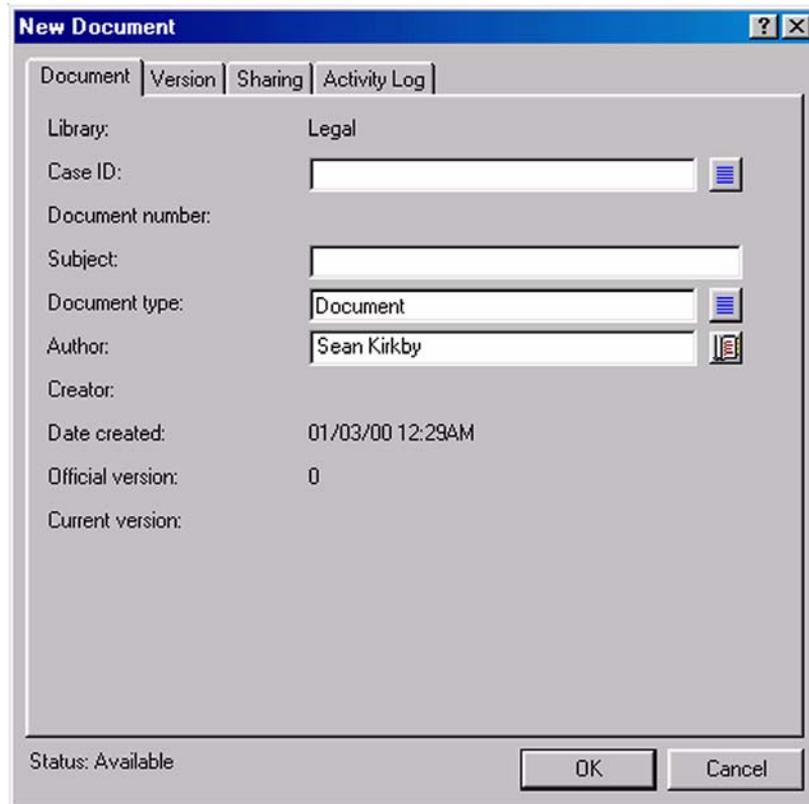


Figure 14: *New Document window.*

The user may type values in free-form fashion, and GroupWise will verify the typed-in value with matching values from the lookup table, or the user may click the selector button which will present a dialog box containing a list of values from the lookup table as shown in Figure 15.

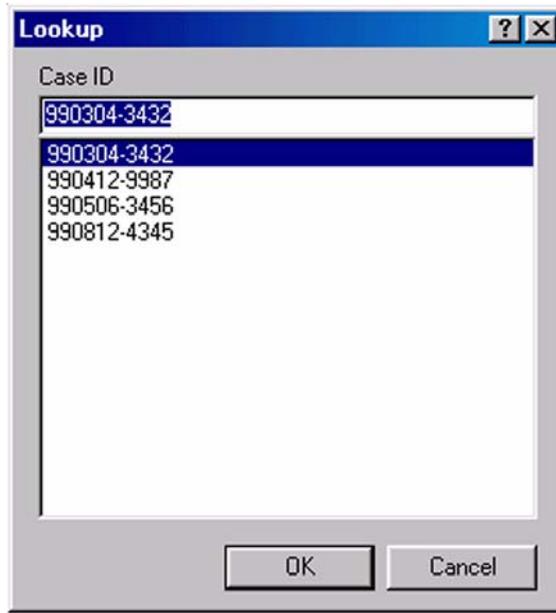


Figure 15: Lookup.

FieldDefinition objects have a Boolean property that indicates whether or not they are associated with a lookup table. Only FieldDefinition objects belonging to a DocumentLibrary.FieldDefinitions collection can be associated with a lookup table.

When a FieldDefinition object is associated with a lookup table, then FieldDefinition.HasLookupTable is TRUE.

In this case, you can call FieldDefinition.GetLookupTable() to get a LookupTableEntries object that contains the lookup table entries for fields based on that FieldDefinition.

```
FieldDefinition.GetLookupTable (Value As {String or Field})
```

This returns a LookupTableEntries collection. This collection has a Count property. Returns a collection of LookupTableEntry objects, each of which has a Value property, which is a variant.

Additional properties of the Field object in GroupWise are:

Property	Data Type	Description
FieldDefinition	FieldDefinition object (R/O)	The field definition object used to create this field. A reference here makes it easy to get reference to the restrictions and methods defined for us.
FieldID	Long (R/O)	Not yet implemented. Field identifier

The LookupTableEntries object is a simple collection object that has a Count property and an Item method. The following code sample shows how to access lookup table entries associated with the Case ID field definition. This sample assumes that there is a list box object called lbTableEntries.

Example in VB:

```
`gwAccount is a valid Account object

` get the library object
gwLegalLibrary = gwAccount.DocumentLibraries.Item( "Legal" )
` get the Case ID field definition object
gwCaseIDFieldDef := gwLegalLibrary.FieldDefinitions.Item( "Case ID", egwString )
` if it has a lookup table...
If gwCaseIDFieldDef.HasLookupTable then
    ` get the collection of table values...
    gwCaseIDValues = gwCaseIDFieldDef.GetLookupTable
    ` clear the list box so we can populate it...
    lbTableEntries.Clear
    ` store the number of entries in the lookup table so the following for..
loop doesn't
    ` have to make a COM call every time (that's too slow!)
    iNumberOfTableEntries = gwCaseIDValues.Count
    ` for each table entry...
    For iCounter = 1 to iNumberOfTableEntries
        // add it to the list box...
        lbTableEntries.Items.Add gwCaseIDValues.Item(iCounter)
    Next iCounter
End If ` if...
```

Example in Delphi:

```
//gwAccount is a valid Account object

// get the library object
gwLegalLibrary := gwAccount.DocumentLibraries.Item( 'Legal' );
// get the Case ID field definition object
gwCaseIDFieldDef := gwLegalLibrary.FieldDefinitions.Item( 'Case ID', egwString );
// if it has a lookup table...
if gwCaseIDFieldDef.HasLookupTable then begin
    // get the collection of table values...
    gwCaseIDValues := gwCaseIDFieldDef.GetLookupTable;
    // clear the list box so we can populate it...
    lbTableEntries.Clear;
    // store the number of entries in the lookup table so the following for..
loop doesn't
    // have to make a COM call every time (that's too slow!)
    iNumberOfTableEntries := gwCaseIDValues.Count;
    // for each table entry...
    for iCounter := 1 to iNumberOfTableEntries do
        // add it to the list box...
        lbTableEntries.Items.Add( gwCaseIDValues.Item( iCounter ) );
    end; // if...
```

The LookupTableEntries object does not have a method that allows you to add values to the table. Adding values to a lookup table can only be done via the GroupWise Administrator or through the GroupWise Admin API. In addition, there are no methods in the Object API that allow the creation of addition lookup tables themselves. This too must be done in the Administrator or through the Admin API.

One final note on lookup tables, in an application that links a custom field to a lookup table and allows users to select a Case ID for each document, it is not a good solution to force the system administrator to be in charge of populating the lookup table. This is where the Admin API comes in. You would probably want to create a custom application that uses the Admin API to allow an end-user to manipulate the values in the Case ID Values lookup table. Otherwise, the end-user would have to use the GroupWise Administrator to do this, and this is certainly not a good solution either!

Related Properties

As you might imagine there could be a large number of case IDs in the Case ID Values lookup table. This could make it difficult for end-users to find the Case ID they are looking for.

It might be beneficial to limit the values that show up in the Case ID Values list to a subset of values based on some other criteria.

For example, you might want the end user to select a geographic location, or a lawyer's name, and have the Case ID Values list show only those Case IDs that pertain to that location or lawyer.

To accomplish this, you create two lookup tables, and designate one of them as being "related" with the other. You would create the table that should be restricted in its display first. In this case, you would create a table called "Lawyers". Then, you would create the "Case ID Values" table. When creating this second table, you would click on the "Related Table" selector, as shown below, which presents a list of already-existing tables to choose from. You would choose the "Lawyers" table.



Figure 16: *Lookup table definition.*

After doing this, each value you add to the Case ID Values lookup table must be associated with an existing value in the Lawyers table. Of course, this means that before you add values to the Case ID Values lookup table, the Lawyers table should already have all possible values in it. As you add values to the Case ID Values table, the dialog box looks like Figure 17.



Figure 17: *Lookup entry.*

Notice the “Parent Value” drop-down list; this will allow you to select a value from the related Lawyers table.

When you select a “Parent Value” for the Case ID Values entry you are creating, you create a special relationship between this Case ID Value and the Lawyer’s table value. When the end-user tries to save a document, the Case ID field will only accept values from the Case ID Values lookup table that match the value selected for the Lawyer field (a custom field created and linked with the Lawyers table).

As an example, look at the GroupWise Administrator’s view of the values in the Case ID Values lookup table.

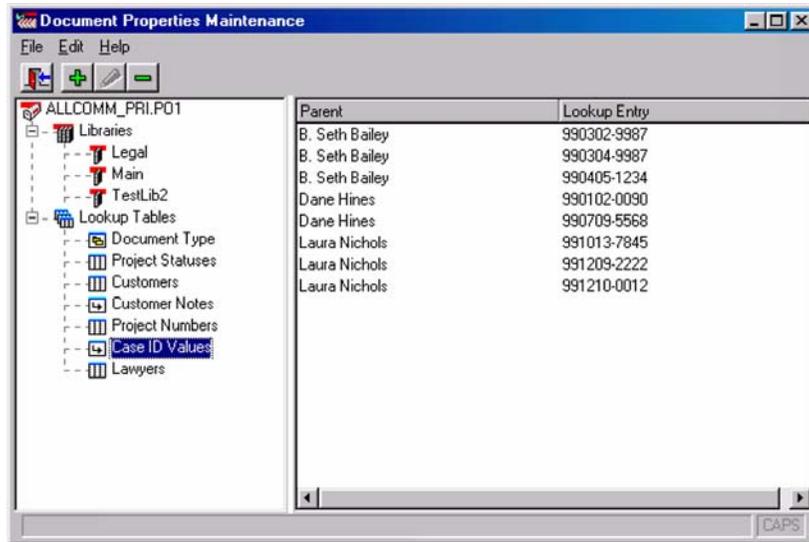


Figure 18: Values in the Case ID Values lookup table.

There are eight values in the table. But when the user tries to save a document and selects one of the lawyers for the “Lawyer” field, the Case ID Values table appears to only have those values that are related to the value of the Lawyer field, as shown in Figure 19.

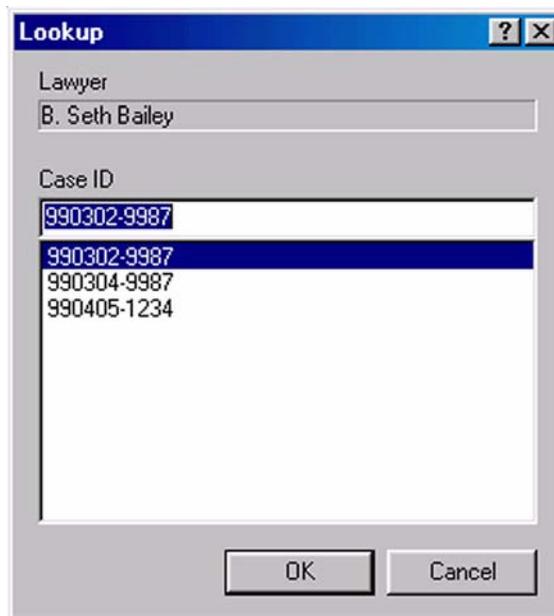


Figure 19: The Case ID Values.

In this case, the Lawyer field for the document being saved is populated with the value “B. Seth Bailey”, and the lookup table for the Case ID field appears to only have those values that are related to B. Seth Bailey.

All FieldDefinition objects have a “RelatedFieldDefinition” property. This property is a FieldDefinition object for a field definition that is the “parent” field for this one. In our example, the FieldDefinition.RelatedFieldDefinition property for the Case ID field would be equal to the FieldDefinition object for the Lawyer field definition. The Lawyer FieldDefinition.RelatedFieldDefinition would be null, or otherwise undefined. It is not a two-way relationship. Only the child field definition has a RelatedFieldDefinition property. For example, see the following code:

Example in VB:

```
`gwAccount is a valid Account object

Set gwLegalLibrary = gwAccount.DocumentLibraries.Item("Legal")
Set gwLawyerFieldDef = gwLegalLibrary.FieldDefinitions.Item("Lawyer", egwString)
Set gwCaseIDFieldDef = gwLegalLibrary.FieldDefinitions.Item("Case ID", egwString)
sLawyerFieldDefName = gwLawyerFieldDef.Name
sCaseIDRelatedFieldDefName = gwCaseIDFieldDef.RelatedFieldDef.Name
```

Example in Delphi:

```
//gwAccount is a valid Account object

gwLegalLibrary := gwAccount.DocumentLibraries.Item('Legal' );
gwLawyerFieldDef := gwLegalLibrary.FieldDefinitions.Item('Lawyer', egwString);
gwCaseIDFieldDef:=gwLegalLibrary.FieldDefinitions.Item('Case ID', egwString);
sLawyerFieldDefName := gwLawyerFieldDef.Name;
sCaseIDRelatedFieldDefName := gwCaseIDFieldDef.RelatedFieldDef.Name;
```

After this code executes, sLawyerFieldDefName = sCaseIDRelatedFieldDefName. However, if we tried to access gwLawyerFieldDef.RelatedFieldDef.Name, we would get an exception since there is no parent field for the Lawyer field.

Understanding how related fields work is important when you are working with lookup table objects.

When you call the FieldDefinition.GetLookupTable() method, you must supply the method with a value. The GroupWise Object API documentation states that this value must either be a string value or a Field object. But this method will also take a numeric value, if the FieldDefinition is of a numeric type and is linked to a lookup table populated with numbers.

If you pass a Field object, it's Value property will be used by the method.

The ultimate value that the `FieldDefinition.GetLookupTable()` method uses is compared with the parent values in the lookup table. The resultant `LookupTableEntries` collection will only contain `LookupTableEntry` objects that have parent values that match the value supplied to `FieldDefinition.GetLookupTable()`. If the lookup table for the `FieldDefinition` object is not a child lookup table, the supplied value is ignored, and the resultant `LookupTableEntries` collection will contain all values in the lookup table.

If the `FieldDefinition` is not related to a lookup table, then the `FieldDefinition.GetLookupTable()` method will either return an empty list or generate an exception.

Summary

In this chapter, you learned to create a custom field for a specific `GroupWise` object like a `Message` or `Folder`, based on an existing `FieldDefinition`. Using the `Object API` or the `GroupWise` client makes it easy to view the list of available pre-defined and custom fields.

In the next chapter, you will learn about `Query` and `Query` related objects.

For more information on `Fields`, please check the `GroupWise Object API` documentation <http://developer.novell.com/ndk/doc.htm>, sample code <http://developer.novell.com/ndk/sample.htm> or visit the `Developer Support Forum` area at <http://developer-forums.novell.com/category/index.tpt>.

Copyright © 2002 by Novell, Inc. All rights reserved.
No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, for any purpose without the express written permission of Novell.

All product names mentioned are trademarks of their respective companies or distributors.

Understanding Filter and Query Related Objects

Chapter 9

Section 1: GroupWise Object API

Filter objects represents saved information in GroupWise while Query objects provide query information and actions. A query object can represent either a stand-alone search which does not persist in the message database or a search associated with a query folder which does persisit in the message database.

Both Filter and Query objects in GroupWise have at their core an expression string which defines the text, numeric, date, enumerated, unary or basic information on which the search or filter is based.

Location identifies the Account, DocumentLibrary or Folder objects that will be associated with a given search.

The following topics are discussed in this chapter.

Contents:

- Filters
- The Filter Object
- Creating Filters
- Creating Queries
- Creating Query Folders
- Locations
- Expressions
- Summary

Filters

Like many other collection objects, Filters have a Count property and Item() method. To access a specific Filter, you must call the Item() method with a single Variant parameter.

```
Filter.Item(Index as Variant)
```

If Index is Long between 1 and Filters.Count, the Filter object at the given Index will be returned. Use a for loop if you don't know the order of filters in the collection. If Index is a long, to avoid an error, it must be between one and Filters.Count. If Index is a string, it will return the Filter object that matches Filter.Name.

The Filter Object

The main purpose of a Filter object is as an argument in a find method. Filters can be saved in an Account.Filters collection. New ones can be created using the Account.Filters.Add() method, and then used as a parameter in one of the following methods:

```
AllMessages.Find()  
Folder.FindMessages()  
MessageList.Find()  
Messages.Find()  
TrashEntries.Find()
```

A notable exception of a method that does not accept a Filter object as a parameter is AddressBookEntries.Find(), which uses a string expression only.

A Filter object has the following properties:

Property	Data Type	Description
Description	String (R/W)	Descriptive text for the filter.
Expression	String (R/W)	The saved filter expression. The syntax of this property should match that used in the Filter dialog box of the GroupWise client.
LastAccessedDate	Date (R/O)	Date the filter was last accessed. This date affects the most recent filters processing in the GroupWise client. The date can be updated by calling the TouchAccessedDate method.
Name	String (R/W)	The name of this filter.

Creating Filters

Use Add() on a Filters collection object to create a new Filter.

```
Filters.Add(Name as String, Expression as String)
```

This method creates a new Filter object using the Name to represent Filter.Name and Filter.Expression as described above. The Filter is added to the existing Filters collection that called it.

Example in VB:

```
`gwAccount is a valid object

Dim gwMessageList As GroupwareTypeLibrary.MessageList
Dim gwFilter As GroupwareTypeLibrary.Filter

Set gwFilter = gwAccount.Filters.Add("MyFilter", _
    "( DELIVERED_DATE >= YESTERDAY )")
Set gwMessageList = gwAccount.MailBox.FindMessages(gwFilter)
```

Deleting Filters and Last Time Accessed

A Filter object only has two methods: Delete() which deletes the filter from its parent collection and TouchAccessDate() which updates the property LastAccessedTime to the current date and time.

Creating Queries

To create a Query or Query Folder, begin with the Account.CreateQuery() method which returns a Query object.

Queries that are created and left standing differ from Queries that are part of a Query Folder in one way:

- The results of a stand-alone Query are returned through the Query.QueryMessages property.
- The results of a Query Folder are returned in the Folder.Messages collection, and the Folder.Query.QueryMessages collection is always empty.

A Query can be executed by itself and return results, or be used to create a Query Folder.

Since the CreateQuery() method takes no arguments, we must fill in the desired properties after the Query object is created.

The properties we are interested in setting before the Query is operational are listed in the table below.

Property	Data Type	Description
Expression	String	The filter expression
Locations	Variant collection	Use the Query.Locations.Add(Location as Variant/Object) method you can tell the Query where to search. Adding just the RootAccount object will search all folders. Otherwise specific Folder objects, Account objects, or DocumentLibrary objects can be added. Folder objects with ObjType of egwQuery or egwCalendar cannot be added.
SearchLocally	Boolean	Affects whether GroupWise Remote executes the Query against the local or master mailbox. Use true if you only want to search the local mailbox and false otherwise.

At this point we can run the query by executing the Query.Start() method.

This method will return before the query completes, and the Query will run asynchronously.

When the Query completes, the Query.Completed flag will be true and Query.QueryResults will contain a MessageList object with the results. In the interim, Query.QueryMessages may return a snapshot of the partial results.

It would be prudent to not run a large query too often. GroupWise provides a Query.CompletionDate property to check the last date and time the Query completed. The Query may already be running when CompletionDate is checked, so check the Completed property first.

GroupWise 5.5 and later also provide a Query.Stop() method to stop a Query.

Creating Query Folders

You can create GroupWise query folders using the Object API. Query folders are Folder objects that execute a query when selected. Only the Folder object with an ObjType = egwQuery, also known as a Query Folder can store a Query object. Items matching the search criteria stored in Query.Expression appear in the Folder. To create a Query Folder, you must first create a Query object.

Once you have a Query object, you should call the CreateFolder() method to create the folder.

```
Query.CreateFolder(FolderName as String,[ParentFolder as Folder])
```

Parameter	Data Type	Description
FolderName	String	Name of the folder
ParentFolder	Folder object	Folder object in which you want to place the new query folder. If omitted, the folder will be created in the root folder

This method returns a `Folder` object that represents the new folder. You may set other folder options for that folder.

On one level, a Query Folder is a Folder where the `ObjType = egwQuery`. In fact all the properties and methods of a Folder work. There is one important difference. Since a Query Folder populates its own Messages collection with references returned by the underlying Query, it cannot contain real objects. It cannot be the destination of a Message or Folder via either the `Messages.Move()`, `Messages.Add()`, or `Folders.Move()` methods. This also means that Folders cannot be nested below a Query Folder.

A Query Folder is a normal Folder, except that its Messages collection can only be populated by the underlying Query.

To perform a query from within the folder, call the `Folder.Query.Start()` method, which will populate the folder object (specifically `Folder.Messages`) with items to be read by the Object API. Note that the GroupWise client will not change unless the user selects the folder. Finally, note that `Folder.Query.QueryMessages` is always empty.

Locations

A Locations collection holds the Accounts, DocumentLibraries, and Folders objects that GroupWise will use as part of a query search.

In addition to the usual Count property and Item() method, Locations also has the following two methods.

```
Locations.Add(Location as Object)
```

Parameter	Data Type	Description
Location	Object	Adds Location to the collection. Location must be an Account object (searches account folders), DocumentLibrary object (searches library document versions), or Folder object (searches folder). If Location is a Folder object, it cannot be a Calendar or Query folder. To search all folders without listing them, search the root account.

```
Locations.Remove(Index as Long)
```

Parameter	Data Type	Description
Index	Long	Removes the location object located at the given Index, from the collection. This method will not delete or destroy the location object. It merely removes the object from the collection.

Expressions

You have to program an Expression about twice in a blue moon. You know you can do it, because you can create the Expression using the client, you just can't seem to decipher the syntax. The client already knows the syntax, so why not let it tell us?

The first rule of writing a Filter Expression is let the client do the work.

So start the client and go to the View->Filter->Edit/Create menu. Go to town, then save the filter using Filter->Save As... on the Filter create interface dialog.

Now let's run the code:

Example in VB:

```
Public Sub ListAllFilters()
Dim gwApplication As GroupwareTypeLibrary.Application
Dim gwAccount As GroupwareTypeLibrary.Account
Dim gwFilters As GroupwareTypeLibrary.Filters
Dim gwFilter As GroupwareTypeLibrary.Filter

Set gwApplication = New GroupwareTypeLibrary.Application
Set gwAccount = gwApplication.Login()
Set gwFilters = gwAccount.Filters
For each gwFilter in gwFilters
    With gwFilter
        Debug.Print
        Debug.Print "Name:", .Name
        Debug.Print "Expression:", .Expression
    End With
End For
End Sub
```

```
End With
Next gwFilter
Set gwFilter = Nothing
Set gwFilters = Nothing
Set gwAccount = Nothing
Set gwApp = Nothing

Exit Sub
```

At this point, you will have to account for quotation marks in the string. If the string requires embedded quotation marks, be sure to provide them in a way your programming language provides for.

Example in VB:

```
` If the Expression should be: ( SUBJECT CONTAINS "Wrox" )
` The statement should be:
gwFilter.Expression = "( SUBJECT CONTAINS ""Wrox"" )"
` note we changed the quotes to a pair of quotes.
```

If we are concatenating the Expression at run time, remember to quote string but not numbers.

Example in VB:

```
Dim sText as String
Dim lNumber as Long
SText = "Hello World"
LNumber = 1
OgwFilter.Expression = "( MESSAGE CONTAINS "" " + sText + "" " ) AND " + _
    "( TOTAL_RECIPIENTS >= " + lNumber + " )"
` The above ends up as:
` ( MESSAGE CONTAINS "Hello World" ) AND ( TOTAL_RECIPIENTS >= 1 )
```

With earlier versions of GroupWise, the Expression property returned by the filter was a user friendly version of the syntactically correct one. Even so, this approach will help you to develop Expressions in record time.

In addition, Filter Expressions can be built by hand.

Text Statements

Text statements allow you to populate MessageList collections based on textual properties that a Message contains. There are many ways to create a basic text statement. The examples below begin with a valid TextField followed by a text operator (MATCHES, CONTAINS, BEGINSWITH) followed by a text constant in quotation:

The following syntax returns objects where the item SUBJECT contains "Internet"

```
(SUBJECT CONTAINS "Internet")
```

The following syntax returns objects where the AUTHOR matches “Mr. Byg”
(AUTHOR MATCHES "Mr. Byg")

The following syntax returns objects where the BodyText of a message begins with “Now hear this”
(MESSAGE BEGINSWITH "Now hear this")

The following syntax returns objects where the subject contains “Inter” OR “Intra” followed by additional characters AND “Test” followed by “Plan” appears also in the subject.
(SUBJECT CONTAINS ("Inter*" OR "Intra?") AND ("Test". . "Plan"))

The syntax * and ? denote wildcard while the syntax “..” requires that the second word follow the first with any number of characters in between.

The following syntax returns objects where FROM matches the string value of the custom field “MY_BOSS”
(FROM MATCHES <MY_BOSS,STRING>)

The following fields are valid for use in any text statement:

ATTACHMENT_LIST

AUTHOR

BC

CALLER_COMPANY

CALLER_NAME

CALLER_PHONE_NUMBER

CC

CLASS_NAME

DOCUMENT_CREATOR

DOCUMENT_FILENAME

DOCUMENT_TYPE

FROM (Matches the FromText property of the Message object. FromText is free-form text, but will always contain the sender’s full name in the text.

FromText does not typically contain the user ID of the sender.)

LIBRARY

MESSAGE

PLACE

RETRIEVED_BY

SUBJECT

TO

VIEWNAME

Numeric Statements

Numeric statements allow you to populate MessageList objects where the collection items contain numeric properties that are less than, greater than, less than or equal to, greater than or equal to, equal to, or not equal to the criteria that you specify. The following examples should help to show you the types of criteria that you can specify.

The following syntax returns objects that have SIZE less than '12000'
(SIZE < 12000)

The following syntax returns objects where all recipients have accepted the message
(NUMBER_ACCEPTED = TOTAL_RECIPIENTS)

The following syntax returns objects where the numeric custom field TOTAL_EMPLOYEES is greater than '50'
(<TOTAL_EMPLOYEES,NUMERIC> > 50)

The following numeric fields are valid for use in any numeric statement:

CURRENT_VERSION_NUMBER
DOCUMENT_NUMBER
NUMBER_ACCEPTED
NUMBER_COMPLETED
NUMBER_DELETED
NUMBER_OPENED
NUMBER_READ
NUMBER_REPLIED
OFFICIAL_VERSION_NUMBER
SIZE
TOTAL_RECIPIENTS
VERSION_NUMBER

Date Statements

Date statements allow you to populate MessageList objects where the collection items contain date properties that are less than, greater than, less than or equal to, greater than or equal to, or equal to criteria that you specify.

When you create a filter, certain constraints on the range of expressions are considered valid. For example, when you specify a relative date, the granularity of a relative offset is the same as the type of pivot point. If you specify TODAY + offset the offset must be in terms of days. THIS_WEEK + offset requires that the offset be in terms of weeks.

The following examples should help to show you the types of criteria that you can specify:

The following syntax returns objects where due date is less than or equal to tomorrow
(DUEEND_DATE <= TOMORROW)

The following syntax returns objects where start date is greater than or equal to Feb 14, 1996 at 8am.
(START_DATE >= 1996/2/14 AT 8:00:00)

The following syntax returns objects where create date is greater than or equal to the current year with offset of 31 additional years.
(CREATE_DATE >= THIS_YEAR 31)

The following syntax returns objects where the custom field BIRTHDAY is equal to this month.
(<BIRTHDAY,DATE> = THIS_MONTH)

The following date fields are valid for use in any date statement:

ASSIGNED_DATE
CREATE_DATE
DELIVERED_DATE
DUEEND_DATE
RETRIEVED_DATE
START_DATE
MODIFY_DATE

Enumerated Statements

Enumerated statements allow you to populate MessageList objects where the collection items contain date properties that are less than, greater than, less than or equal to, greater than or equal to, or equal to criteria that you specify. The following examples should help to show you the types of criteria that you can specify:

The following syntax returns objects where the message priority is high
(PRIORITY = HIGH)

NORMAL and LOW are also valid priorities.

The following syntax returns objects where attachment type is not equal to OLE
(ATTACHMENT_TYPE <> OLE)

Other valid attachment types include: APPOINTMENT, DOCPREFERENCE, FILE, MAIL, MESSAGE, MOVIE, NOTE, PHONE_MESSAGE, SOUND, TASK

The following syntax returns objects where box type is incoming
(BOX_TYPE = INCOMING)

Other valid box types are: OUTGOING, PERSONAL, DRAFT

Unary Statements

Unary statements allow you to populate MessageList objects where the collection items contain unary properties such as Type and Status equal the criteria you specify. The following examples should help to show you the types of criteria that you can specify.

The following returns objects type is Mail or Appointment
(MAIL OR APPOINTMENT)

Other valid types include: DOCREFERENCE, NOTE, TASK,
PHONE_MESSAGE,

Type statements are mutually exclusive. A message object may have only one type.

The following returns objects where Accepted and Completed is TRUE
(ACCEPTED AND COMPLETED)

The following returns objects where type is Document, status is not hidden and not read.
(DOCUMENT AND NOT HIDDEN AND NOT READ)

Additional statuses are: OPENED, READ, HIDDEN, DELEGATED,
DELIVERED, ON_CALENDAR,
OFFICIAL_DOCUMENT_VERSIONS_ONLY, PRIVATE,
REPLY_REQUESTED, ROUTED, DOCVERSION_CHECKED_OUT,
DOCVERSION_IN_USE, DOCVERSION_CONNECTED_READ_WRITE,
DOCVERSION_ARCHIVED, SEARCH_AS_LIBRARIAN

Status statements are not mutually exclusive. Message objects may have any of the statuses.

Basic Expressions

Use the following syntax to identify certain message types in a Query or Filter Expression. Any statement, Numeric, Text, Date, Enumerated, Unary may be connected with either an AND or OR into a compound statement. Parentheses are required and may only be used at the group level.

Returns only Mail objects.

(Mail)

Returns Mail or Appointment objects.

(Mail) or (Appointment)

Summary

In this chapter you learned about GroupWise custom fields; how to create, access, and delete them. We covered Query objects and folders and also Filter Expression syntax which is used to define the MessageList that is returned based on text, numeric, date, and other criteria.

For more information on Filter and Query objects, please check the GroupWise Object API documentation <http://developer.novell.com/ndk/doc.htm>, sample code <http://developer.novell.com/ndk/sample.htm> or visit the Developer Support Forum area at <http://developer-forums.novell.com/category/index.tpt>.

Copyright © 2002 by Novell, Inc. All rights reserved.
No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, for any purpose without the express written permission of Novell.

All product names mentioned are trademarks of their respective companies or distributors.

Overview

Chapter 16

Section 3: GroupWise C3POs

Custom Third-Party Objects (C3POs) extend the functionality of GroupWise as follows:

- Add menus, menu items, menu actions, and separators to the client
- Add buttons to the toolbar
- Trap predefined commands
- Catch certain GroupWise events
- Build custom message types with custom message icons

In addition, C3POs are often used with other APIs such as the GroupWise Object API, Administrative Object API, and Token API to perform some action.

When GroupWise begins, it searches the Windows registry for information about the C3POs that should be loaded along with the client. Keys in the registry point to a COM server objects that you have built into a .EXE or .DLL. This is the file that is your C3PO. GroupWise uses a C3PO Manager to interface with your C3PO which must expose certain methods and properties in order to be understood by GroupWise.

Although it is possible to build the COM objects that the Manager is looking for on your own, Novell includes a C3PO Creation Wizard along with the documentation and sample that makes the creation of a C3PO extremely easy. The wizard walks you through a series of menus, prompting you for information on the type of component that you'd like to create; ultimately the wizard produces the following files in Visual Basic, Delphi, or C++.

- C3POServer
- CommandFactory
- GWCommand
- EventMonitor

- IconFactory

These files may be used, or not used, depending on what you have coded the C3PO to do.

C3POServer

The C3POServer object is the only object required by the C3PO Manager. It is used by the Manager to get at the other objects of your C3PO. This object defines the following four properties and three methods:

Properties	<ul style="list-style-type: none"> • <i>Description</i> contains a short description of the C3PO. • <i>CommandFactory</i> returns an object if your C3PO add menus, menu items, buttons or take over predefined commands; otherwise the property is NULL. • <i>EventMonitor</i> returns an object if your C3PO handles events such as OnReady, OnShutdown, OnDelivery, and OnOverflow. Otherwise, like CommandFactory, this property is NULL. • <i>IconFactory</i> returns an object if your C3PO builds custom messages, otherwise is NULL.
Methods	<ul style="list-style-type: none"> • <i>CanShutdown</i> is called when GroupWise needs to shutdown. If TRUE is returned, the client calls Delnit and shuts down. If FALSE is returned, the manager will check again and again until TRUE is finally returned. • <i>Delnit</i> is called by the Manager to release any holds that exist on the object. • <i>Init</i> is passed a Manager object by GroupWise to act as the medium between the C3PO and the client.

Although the wizard will write all the code needed to create an object of type C3POServer, it is useful to understand the methods and properties that this object makes available.

CommandFactory

The CommandFactory object is used to handle predefined GroupWise commands such as when a mail message is opened or sent. This object is also used when you want to add custom menus and toolbar buttons to the client.

Properties	None
Methods	<ul style="list-style-type: none">• <i>BuildCommand</i> creates a GWCommand object.• <i>CustomizeContextMenu</i> adds new menu items and menu actions to the GroupWise context menus that appear when a user right clicks on a message object in the client.• <i>CustomizeMenu</i> adds new menu items and menu actions to existing GroupWise menus.• <i>CustomizeToolbar</i> adds buttons to the toolbar.• <i>Init</i> returns information to the Manager that identifies whether menus, toolbars or predefined commands will be customized. This is the first method called in CommandFactory.• <i>WantCommand</i> determines whether a predefined command is supported. If, for example, you wish to perform some action whenever a user creates a new message, the oncompose command will be listed in this method as one you want to handle.

CommandFactory is returned to GroupWise via the CommandFactory property of the C3POServer object.

GWCommand

The GWCommand object defines an instance of a command. This object contains an Execute method that is called when a user presses a menu, selects a toolbar button or has chosen to catch and handle a predefined command.

Properties	<ul style="list-style-type: none">• <i>BaseCmd</i> stores the default functionality of a GroupWise command. This property is often used to continue a process after your C3PO has had a chance to perform some custom action.• <i>LongPrompt</i> displays text on a menu.• <i>PersistentID</i> is a string value that identifies the C3PO to GroupWise. If multiple C3POs are loaded this value, which is set to 0 by default, should be changed to some other value to avoid confusion by the client when making a request.• <i>ToolTip</i> displays text on the toolbar.
Methods	<ul style="list-style-type: none">• <i>Execute</i> determines actions to execute as part of a C3POs program flow. This is the place to put custom code that will run when a user selects a menu or toolbar or when a predefined command is trapped.• <i>Help</i> is not used by the C3POManager but may be used by another C3PO.• <i>Undo</i> like help, is not used by the C3POManager but may be used by another C3PO.• <i>Validate</i> checks or sets the state of a button or menu item based on some criteria such as whether the message is new.

EventMonitor

The EventMonitor object is used to handle one of four GroupWise events.

Properties	None
Methods	<p><i>Notify</i> receives one of the following four low level actions along with a context string.</p> <ul style="list-style-type: none">• eGW_EVT_DELIVERY is passed to Notify when GroupWise receives a new item.• eGW_EVT_READY lets your C3PO know that the client has been initialized.• eGW_EVT_SHUTDOWN lets your C3PO know that GroupWise is about to shutdown.• eGW_EVT_OVERFLOW is passed when too much information is returned at once.

When the context and action you want to handle matches the context and action that is passed to Notify, custom code that you place in Notify will run.

IconFactory

The IconFactory object is used when you build a C3PO to create or handle custom message types.

Properties	None
Methods	<ul style="list-style-type: none">• GetIcons returns a file that contains icon information that your C3PO will use with a custom message type.

Flow of Control

It is helpful to understand that way in which GroupWise goes about adding menus, toolbars, catching commands, events, and creating custom message types.

Initialize

GroupWise calls Init() in CommandFactory to identify the items that a C3PO plans to handle e.g. menus, context menus, toolbars, or predefined commands (described in the next section). Possible flags are:

```
eGW_CMDINIT_MENU  
eGW_CMDINIT_CONTEXT_MENU  
eGW_CMDINIT_TOOLBARS  
eGW_CMDINIT_NO_PREDEFINED
```

A combination of `eGW_CMDINIT_MENUS` + `eGW_CMDINIT_TOOLBARS` tells the Manager that your C3PO should be called before GroupWise draws a client menu or toolbar. `eGW_CMDINIT_CONTEXT_MENUS` tells the Manager that your C3PO should only be called when a user right clicks on an object or a context menu is about to be drawn.

Customize

Menus. `CustomizeMenu()` is called if your C3PO has stated in `Init()` that it will modify GroupWise menus. A menu object and context are passed to the method. If your C3PO has code to handle menus with a context of `'GW.MESSAGE'` and this is the context that the method receives, your C3PO will have the chance to modify the menu of the GroupWise message object that will be drawn before it is shown to the user. This gives your C3PO the chance to remove, override, or add menu items.

Likewise, if your C3PO requests to handle context menus, `CustomizeContextMenu()` is called and provides you an opportunity to customize the menu before it is shown to a user.

Toolbars. The `CustomizeToolBar` method is called with a toolbar object and a context before GroupWise adds a button to a toolbar. If the context passed to this method matches the context that your C3PO is interested in handling, the method runs code to add the button which will show up on the Calendar, FindResults or whatever window context matches the criteria you have set.

While not all of the system buttons can be removed, it is possible to remove some and easy to add new buttons as described later on. Adding a toolbar button requires that you set the file path in the `CustomizeToolBar` method to point to a valid icon file. If this file is missing or invalid, your button will not display.

`eGW_CMDINIT_NO_PREDEFINED` lets the Manager know that your C3PO wants to hear about commands that occur in the GroupWise client. Based on the value returned by `Init()`, the appropriate customize or command method is then called next.

In addition to setting the menu, context menu or toolbar appearance, each customize method also creates a `GWCommand` object which it associates with a specific action e.g. a user selects a menu or toolbar button or opens a new mail message.

Validate

The call that GroupWise makes to `Validate` gives you the chance to set or check the state of a GroupWise menu or toolbar button. You may, for example, want to give users the opportunity to select a certain menu if a message happens to be new. For additional information on how `Validate` can work, please see Chapter 19: Putting It All Together.

Execute

Following the call to Validate, GroupWise calls Execute to determine what menu or button has been selected or what predefined command has been caught. The appropriate response is invoked based on the actions you have chosen to run. It is within the Execute method that your C3PO should place custom code to use the GroupWise Object, Administrative, or Token API.

Trapping Predefined Commands

Similar in program flow to that of menus and toolbars, a call to Initialize that does *not* return eGW_CMDINIT_NO_PREDEFINED is followed by a call from the C3PO Manager to WantCommand. If this method returns TRUE meaning that the C3PO has matched the context and command passed to the method with the context and command that you have decided your C3PO will handle, a call to BuildCommand is made. BuildCommand creates a GWCommand object, sets the ID and saves the base command of the action that would have been performed for later use. The command object is then returned to the Manager. Execute is then called to run custom code in place of the trapped command. BaseCmd is often used after a C3PO has run through some custom execution to invoke the default GroupWise action.

Catching GroupWise Events

Program flow for handling events is very simple because the C3PO Manager calls only the Notify method of the EventMonitor class. All custom handling that should take place when one of the events (above) is performed in the context of Notify.

Once you've written code to stand behind a menu, toolbar, command, event, or custom message, it is important that you compile your C3PO to a proper .EXE or .DLL file and register the file with Windows which will enable GroupWise to find and load the C3PO.

Registration

On startup, GroupWise searches the Windows registry for information on the C3POs that it should load. A C3PO is "registered" when proper keys are added to the registry creating an association between the C3POServer and a particular name type. GroupWise looks for C3POs in the registry as follows:

```
HKEY_LOCAL_MACHINES\Software\Novell\GroupWise\5.0\C3PO\Datatypes\
```

From this location, GroupWise searches 'GW.MESSAGE', 'GW.CLIENT' or a subtype therein e.g. 'GW.MESSAGE.XXX' or 'GW.CLIENT.XXX' for references to the C3POs that should be loaded.

The following contexts are associated with GW.MESSAGE

GW.MESSAGE - identifies any of the following message subtypes

GW.MESSAGE.MAIL - identifies a GroupWise Mail message

GW.MESSAGE.MAIL.Internet - identifies a GroupWise mail message from the internet

GW.MESSAGE.MAIL.NGW.DISCUSS - identifies a GroupWise Shared Notification

GW.MESSAGE.APPOINTMENT - identifies a GroupWise Appointment

GW.MESSAGE.TASK - identifies a GroupWise Task

GW.MESSAGE.NOTE - identifies a GroupWise Note

GW.MESSAGE.PHONE - identifies a GroupWise PhoneMessage

GW.MESSAGE.DOCREF - identifies a GroupWise DocumentReference

In addition, it is possible to create custom subtypes of a message by appending text to an existing message context. For example, your C3PO may wish to handle or create custom messages of type "GW.MESSAGE.MAIL.NewType". By creating or checking for this special context, your C3PO can identify a message of NewType and your C3PO can take appropriate action.

Unlike the root above, the second type of context "GW.CLIENT" cannot be extended. Instead, GroupWise understands the following client contexts.

GW.CLIENT - identifies any of the following client windows

GW.CLIENT.WINDOW.ATTACHVIEWER - identifies the attachment viewer

GW.CLIENT.WINDOW.BROWSER - identifies the browser window

GW.CLIENT.WINDOW.CALENDAR - identifies all calendar views

GW.CLIENT.WINDOW.DOCUMENTLIST - identifies the Document list window.

GW.CLIENT.WINDOW.FINDRESULTS - identifies the Query results window.

GW.CLIENT.WINDOW.PROPERTIES - identifies the Properties window

GW.CLIENT.WINDOW.QUICKVIEWER - identifies the Quickviewer window

Additional keys are placed in the Windows registry to clarify a C3POs use. Under each key, you will find an Object key that identifies the objects supported by the C3PO and an Event key that lists events that the C3PO is interested in intercepting.

How to Register

Although you can register a C3PO by hand (just add the required keys to the registry directly) a C3PO created with the wizard contains code that will register the C3PO for you. From Start | Run type the location path to your C3PO or drag and drop the .EXE or .DLL and enclose the path in quote marks followed by /r. It is important that the enclosed string not contain /r and not the other way around.

Incorrect

```
"C:\WINDOWS\Desktop\sampleC3PO\sampleC3PO.exe /r"
```

Correct

```
"C:\WINDOWS\Desktop\sampleC3PO\sampleC3PO.exe" /r
```

Check the registry to make sure that the proper keys were added. It is also possible to register your C3PO from the command line.

How to Unregister

To unregister a C3PO, the same process that you follow to register your C3PO should be followed with the following change that you use /u rather than /r. Alternatively, you can just delete the associated keys from the registry.

To make this section more readable, we'll abbreviate the following:

```
HKEY_LOCAL_MACHINE\software\Novell\Groupwise\5.0\C3PO\DataTypes\  
to:
```

```
...DataTypes\  
The C3PO class name will be the same as the project name. The C3POServer object may be renamed, but usually is not. The generic form of the registry keys are:
```

```
...DataTypes\<<Context>\<CLSID>.<C3POServer object name>\Objects  
and
```

```
...DataTypes\<<Context>\<CLSID>.<C3POServer object name>\Events
```

```
The values entered under these keys depend on the functionality you wish the C3PO to have. Here are the keys for a HelpDesk project.
```

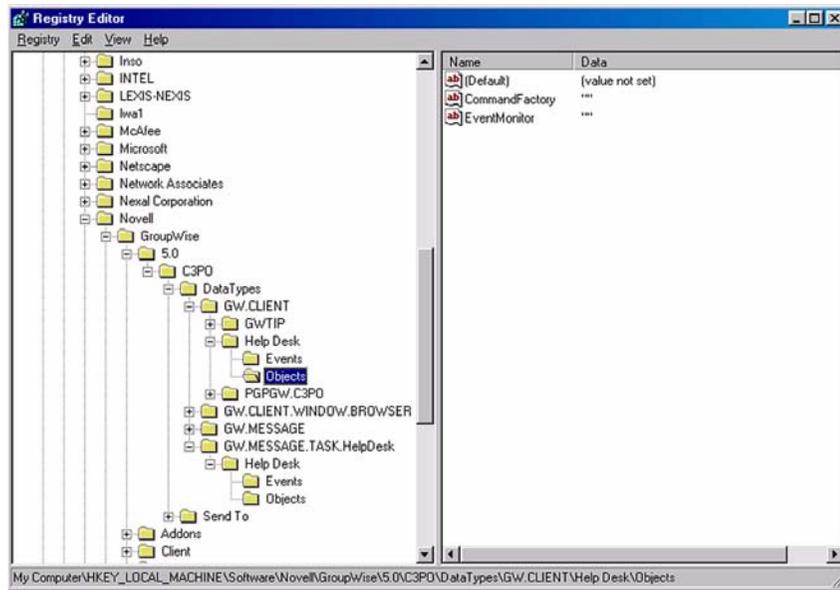


Figure 1: Registry Editor.

VB Example

```
Public Sub RegC3po()

    Dim lResult As Long
    Const HKEY_LOCAL_MACHINE = &H80000002
    Const REG_SZ = 1
    Dim btBuffer As Byte
    Dim sServerKey As String

    btBuffer = 0

    sServerKey="SOFTWARE\Novell\GroupWise\5.0\C3PO\DataTypes\GW.MESSAGE.TASK.HelpDesk
k
\HelpDesk.C3POServer"
    Call RegCreateKey(HKEY_LOCAL_MACHINE, sServerKey + "\Objects", lResult)
    Call RegSetValueEx(lResult, "IconFactory", 0, REG_SZ, btBuffer, 0)
    Call RegCreateKey(HKEY_LOCAL_MACHINE, sServerKey + "\Events", lResult)

End Sub
```

Delphi Example

```
function RegisterServer : HRESULT;
var
    Reg : TRegistry;
    sRegKeyName : string [120];
    sAppName : string [120];

begin
```

```

sAppName := 'Help Desk';
Reg := TRegistry.Create;
Reg.RootKey := HKEY_LOCAL_MACHINE;
sRegKeyName := '\SOFTWARE\Novell\GroupWise\5.0\C3PO\DataTypes\GW.CLIENT\Help
                Desk';
Reg.OpenKey (sRegKeyName, TRUE);
Reg.OpenKey ('Objects', TRUE);
Reg.WriteString ('EventMonitor', '');
Reg.OpenKey (sRegKeyName, TRUE);
Reg.OpenKey ('Events', TRUE);
Reg.WriteString ('OnShutdown', '');

sRegKeyName :=
'\SOFTWARE\Novell\GroupWise\5.0\C3PO\DataTypes\GW.MESSAGE.TASK.HelpDesk\Help
Desk';
Reg.OpenKey (sRegKeyName, TRUE);
Reg.OpenKey ('Objects', TRUE);
Reg.WriteString ('EventMonitor', '');
Reg.OpenKey (sRegKeyName, TRUE);
Reg.OpenKey ('Events', TRUE);
Reg.WriteString ('OnDelivery', '');
sRegKeyName :=
'\SOFTWARE\Novell\GroupWise\5.0\C3PO\DataTypes\GW.MESSAGE.TASK.HelpDesk\Help
Desk';
Reg.OpenKey (sRegKeyName, TRUE);
Reg.OpenKey ('Objects', TRUE);
Reg.WriteString ('IconFactory', '');
Reg.OpenKey (sRegKeyName, TRUE);
Reg.OpenKey ('Events', TRUE);

sRegKeyName := '\SOFTWARE\Novell\GroupWise\5.0\C3PO\DataTypes\GW.CLIENT\Help
Desk';
Reg.OpenKey (sRegKeyName, TRUE);
Reg.OpenKey ('Objects', TRUE);
Reg.WriteString ('CommandFactory', '');
Reg.OpenKey (sRegKeyName, TRUE);
Reg.OpenKey ('Events', TRUE);

sRegKeyName := '\SOFTWARE\Novell\GroupWise\5.0\C3PO\DataTypes\GW.CLIENT\Help
Desk';
Reg.OpenKey (sRegKeyName, TRUE);
Reg.OpenKey ('Objects', TRUE);
Reg.WriteString ('CommandFactory', '');
Reg.OpenKey (sRegKeyName, TRUE);
Reg.OpenKey ('Events', TRUE);

sRegKeyName :=
'\SOFTWARE\Novell\GroupWise\5.0\C3PO\DataTypes\GW.MESSAGE.TASK.HelpDesk\Help
Desk';
Reg.OpenKey (sRegKeyName, TRUE);
Reg.OpenKey ('Objects', TRUE);
Reg.WriteString ('CommandFactory', '');
Reg.OpenKey (sRegKeyName, TRUE);
Reg.OpenKey ('Events', TRUE);
Result := S_OK;
end;

```

In this chapter, we looked briefly at the classes, methods, and properties that make up a C3PO. Subsequent chapters will walk you through the full process of adding menus, context menus, toolbar buttons to the client, capturing commands, events, and creating custom message types. The final step, after you have developed your C3PO, is to register it with Windows.

Copyright © 2002 by Novell, Inc. All rights reserved.
No part of this document may be reproduced or transmitted
in any form or by any means, electronic or mechanical,
including photocopying and recording, for any purpose
without the express written permission of Novell.

All product names mentioned are trademarks of
their respective companies or distributors.

Customizing Menus and Toolbars

Chapter 17

Section 3: GroupWise C3POs

GroupWise uses the methods of `CommandFactory` along with the `GWMenu` object to modify menus in GroupWise.

Customizing Main Menus

`CommandFactory.CustomizeMenu()` takes a GroupWise context and menu object.

```
CustomizeMenu(Context ;GWMenu)
```

Name	Data Type	Description
Context	String	A string containing the current type of window being built by GroupWise. You will want to test the value of Context against the window type of the menu you want to modify, and if the values match, you should take steps to modify the menu.
GWMenu	GWMenu Object	GWMenu that contains all of the information about the menus and that allows you to modify your menus.

Because a C3PO object may be subclassed, you need to test for the highest level object upon which you want to act. If, however, you do an “equals” test for the high level object, then the code will fail most of the time. For example, if you write:

```
If Context = 'GW.MESSAGE'
```

Your code will not be triggered for `'GW.MESSAGE.MAIL'` objects. Instead, you should do a compare on the first part of the `Context` like this:

```
If (CompareText('GW.MESSAGE',Copy(context,1,10)) = 0)
```

Alternatively, the Novell sample code uses:

```
If Pos('GW.MESSAGE', Context) <> 0 then
```

To actually define a menu to customize, you need to first tell the C3PO that you want to customize menus.

VB Example

```
function CommandFactory.Init(lcid : longint): longint;

begin
    result := eGW_CMDINIT_MENUS;
end;
```

Delphi Example

```
function CommandFactory.Init(lcid : longint): longint;

begin
    result := eGW_CMDINIT_MENUS;
end;
```

Then, when `CommandFactory.CustomizeMenu` is called, you can test whether the `Context` matches the item you want to customize. The following samples test for 'GW.CLIENT.WINDOW.BROWSER' will affect menus that appear on the main `GroupWise` window.

VB Example

```
Public Function CustomizeMenu(sGWContext As String, objGWMenu As Object) As Boolean

    Dim Menu As Object

    If sGWContext = "GW.CLIENT.WINDOW.BROWSER" Then           ' Check for correct
        context Set Menu = objGWMenu                          ' Get Main menu object
    End If
End Function
```

Delphi Example

```
function CommandFactory.CustomizeMenu( Context: string;
                                       GWMenu: variant): ToleBool;

var
    vMenuItems : variant;
    vMenu : variant;
    vSeparator: variant;
    Cmd : Command;
```

```

begin
    If CompareStr(Context, 'GW.CLIENT.WINDOW.BROWSER') = 0 then //Check for
correct context
    begin
        end;

        result := FALSE;
end;

```

We can't go any further until we discuss the `GWMenu` object.

The `GWMenu` Object

`GWMenu` represents an entire menu list on which all menu modification actions take place. The `GWMenu` parameter passed into `CustomizeMenu()` represents the entire menu bar for the window that is being created. This menu bar will hold many sub-menus (such as File, Edit, etc.) which themselves are `GWMenu` objects.

The key properties and methods of this object are listed in the following table.

Name	Data Type	Description
Caption	String	This property holds the displayed name of the menu. Note that if you put an ampersand (&) before any letter, that letter will be underlined and constitute a hot-key for the menu.
MenuItems	<code>GWMenuItems</code> Object	This collection property will hold each of the sub-menus and menu commands (or "actions") of the particular menu. For the <code>GWMenu</code> that is passed into <code>CustomizeMenu()</code> , this will contain the File, Edit, etc., menus. You will do most of your work on this property when you create new menus and new menu items.
<code>GWCommand</code>	<code>GWCommand</code> Object	This is the object that represents the actual function of the menu. It is of type <code>GWCommand</code> . See Chapter 16: <code>GWCommand</code> for a discussion of this object. You will want to set this property, even though menus don't really take actions in <code>GroupWise</code> , if you want to validate the menu. That is, if there is ever a time when you want an entire custom menu to be dimmed out, you should set a command for the menu. Otherwise the command's <code>Validate()</code> method will never be called.
Parent	<code>GWMenu</code> Object	This object property is a reference to the <code>GWMenu</code> object that contains the current <code>GWMenu</code> . This is helpful for moving up and down the menu hierarchy.
ObjType	Enumerated Integer	Check this enumerated property to ensure that you are working with a <code>GWMenu</code> object rather than a <code>GWMenuItem</code> object (discussed below), as both objects are subtypes of the <code>GWMenuItem</code> object. The <code>GWMenu</code> object will have a value of <code>**</code> .

Name	Data Type	Description
IsModified	Boolean	This boolean property will be true if this menu has been modified by any C3PO, and false otherwise.

The GWMenu object has one key method which removes a menu.

Delete()

You cannot delete the top-level menu in a window – thus, this method will fail if you attempt to call it on the GWMenu that is passed into CustomizeMenu().

Let's get a reference to the File | New menu and we will show you how to use the object later.

VB Example

```
Public Function CustomizeMenu(sGWContext As String, objGWMenu As Object) As Boolean

    Dim Menu As Object

    If sGWContext = "GW.CLIENT.WINDOW.BROWSER" Then          ' Check for correct
        context Set Menu = objGWMenu                        ' Get Main menu object
    End If
End Function
```

Delphi Example

```
function CommandFactory.CustomizeMenu( Context: string;
                                       GWMenu: variant): ToleBool;

var
  vMenuItems : variant;
  vMenu : variant;
  vSeparator: variant;
  Cmd : Command;
begin

    If CompareStr(Context, 'GW.CLIENT.WINDOW.BROWSER') = 0 then //Check for
correct context
    begin
        vMenu := GWMenu; // get menu from GWMenu
    end;

    result := FALSE;
S
end;
```

The GWMenuItem Object

The `GWMenuItem` object is nearly exactly the same as the `GWMenuItem` object because both objects are subtypes of the `GWMenuItem`. The only differences are that 1) `GWMenuItem.ObjType` is a `MenuItem` in a `GWMenuItem` and 2) The `GWMenuItem.MenuID` property defines the unique ID of a menu action. In practice, the difference is that a `GWMenuItem` corresponds to an item on a menu that actually performs some action in GroupWise but does not have any subitems like menus. Thus, `GWMenuItem` does not have a `MenuItems` property. In order to make a `GWMenuItem` actually take effect, you have to set the `GWCommand` property. See the section below on adding new menus and menu items for an in-depth discussion of associating commands with menu items.

The GWMenuItems Collection Object

When you obtain a reference to the `GWMenuItem.MenuItems` property, you are working with an object of type `GWMenuItems`. This collection object is similar to each of the other collection items in GroupWise. The key properties of `GWMenuItems` are:

Name	Data Type	Description
Count	Unsigned Integer	Like other collection objects, this property holds an integer that equals the number of items in the collection.
Parent	GWMenuItem Object	This object property is a reference to the <code>GWMenuItem</code> object that contains this <code>GWMenuItems</code> collection. This is helpful for moving up and down the menu hierarchy.

Like many other collection objects, `Item()` is used to access individual objects from the collection.

`Item(Index)`

Name	Data Type	Description
Index	VARIANT	If <code>Index</code> is an integer between 1 and <code>Count</code> , then the <code>GWMenuItem</code> corresponding to that location will be returned. This is useful for iterating through each item in the collection. Alternatively <code>Index</code> can be a string that is the display name of the menu you wish to retrieve. This can be a bit touchy, as the display name may have an ampersand or not be what you expect if another version of GroupWise with another language than what you anticipate is used.

This method returns the `GWMenuItem` (which will be either the `GWMenuItem` subclass or the `GWMenuItem` subclass) corresponding to the value of the single variant parameter called `Index`. Look at the returned item's `ObjType` property to determine what has been returned by the method.

Unlike other collection objects, `GWMenuItem`s also has two other methods to obtain references to the items held by the collection. The first is `FindByHMenu()`.

```
FindByHMenu(hMenu)
```

This method returns the `GWMenu` that has a `GWCommand` property with a specific `Menu` handle. This method takes a single parameter called `hMenu`, which is a long integer corresponding to the Windows API handle of the desired menu item. To obtain the handle of an item for future use, you may have to walk the menubar using the Windows API. This method is recursive – it will move down levels to find the menu item.

The second additional method is `FindByID()`.

```
FindByID(ID)
```

Use this method when you know the actual ID (not the string `PersistentID`) of a menu item. This method will return a `GWMenuItem`. This method takes a single variant parameter called `ID`. To get the ID of the menu you desire, you should keep track of the `GWMenuItem.MenuID` property. This method is also recursive – it will search through each of the levels of menus to find the menu item with the particular ID.

Let's now access the menu items for a specific menu so that we can begin to manipulate the items.

VB Example

```
Public Function CustomizeMenu(sGWContext As String, objGWMenu As Object) As Boolean

    Dim Menu As Object

    If sGWContext = "GW.CLIENT.WINDOW.BROWSER" Then          ' Check for correct
        context
        Set Menu = objGWMenu                                ' Get Main menu object
        Set Menu = Menu.MenuItems.Item("File")              ' get menu File
        Set Menu = Menu.MenuItems.Item("New")              ' get menu New
    End If
End Function
```

Delphi Example

```
function CommandFactory.CustomizeMenu( Context: string;
                                        GWMenu: variant): ToleBool;

var
    vMenuItems : variant;
```

```

vMenu : variant;
vSeparator: variant;
Cmd : Command;
begin

    If CompareStr(Context, 'GW.CLIENT.WINDOW.BROWSER') = 0 then //Check for
correct context
    begin
        vMenu := GWMenu; // get menu from GWMenu
        vMenu := vMenu.MenuItems.Item('File'); //get menu File
        vMenu := vMenu.MenuItems.Item('New'); //get menu New
    end;

    result := FALSE;

end;

```

Add Menus and MenuActions

When `CustomizeMenu()` method is called, you modify the `MenuItems` property of the `GWMenu` object that is passed in as a parameter. You do not modify `GWMenu` itself because this is the top level container. For example, the File menu is usually the first menu item in the `GWMenu` object.

From this point, you can add either a new menu or a new menu action. For user interface reasons, you typically would not add a menu action to the top-level menu – users just don't expect to have a top level item result in instant action. It is much better to add a new menu to the top level menu, and then add the menu action to that menu. You will use the following methods of `GWMenuItems` to configure menus:

`AddMenu()`: This method adds a new menu to the items collection.

`Add()`: This method will add a new menuaction item in the given set of menu items.

`AddSeparator()`: This method adds a menu separator (one of those lines you see in a menu to separate the items) within the menu items.

The AddMenu Method

The `AddMenu()` method takes one required and two optional parameters.

```
AddMenu(Caption[,Item][,AddBefore])
```

Name	Data Type	Description
Caption	String	This is the string that the user sees on the menu. Add an ampersand (&) before any single character to make that character an underlined hot-key.

Name	Data Type	Description
CommandItem	GWCommand Object	This is an optional GWCommand object. You create this object using GWCommand.Create(). While a menu does not execute any commands, you might still want to attach a command to the menu for searching purposes, for validating the menu, and for assigning a long prompt to the command.
AddBefore	Variant	If AddBefore is an integer, then the menu will be added before that item. The integer must be less than MenuItem.Count. If AddBefore is a string, then the new menu will be added before the menu with that display name. This is important, because you can always tie a menu to another menu, even if you don't know where the menu is. The ampersand (&) is ignored in the display name. If you omit AddBefore, the menu will be added to the end of the current menu items.

AddMenu () returns the new GWMenu object, which you should assign to a variable to manipulate.

The Add Method

The Add () method takes the same parameters as the AddMenu () method, except that CommandItem is no longer optional.

```
Add(Caption,CommandItem[,AddBefore])
```

All GWMenuItem items have a GWCommand associated with them. The GWCommand.Execute method is called when the menu action item is activated.

The AddSeparator Method

This takes a single optional parameter, AddBefore, which works just like the AddBefore parameters associated with the AddMenu () and Add () parameters.

```
AddSeparator([AddBefore])
```

Set the Menu or MenuItem Properties

Once you have obtained the new menu or menu action item, you may set the various properties of the menu, depending on what you want to achieve.

You can also obtain a reference to a submenu and then manipulate the sub menu just as you would the top-level menu. You can even add a new menu to any menu, creating a hierarchical menu structure – much like the File | New menu in GroupWise.

Let's now take a look at actually adding some menus and menu items using all of these methods. Note in the following that several variants are declared – this is the output from the C3PO wizard. Note that these command variants actually need not be redeclared; if they were released after use, they could be reused. Likewise, they are shown here as local variables, while the Novell samples usually show global variables. This is a matter of taste, and will depend on whether you want to access the commands at a later time.

The following code does the following three things in the browser: 1) Adds a “HelpDesk” menu item to the “File | New” menu; 2) Creates a “Sample” menu under the tools menu; and 3) Adds a “Sample Item” menu item to the “Sample” menu that just got created. See if you can spot each of the new menus.

VB Example

```
Public Function CustomizeMenu(sGWContext As String, objGWMenu As Object) As
    Boolean

    Dim Menu As Object

    If sGWContext = "GW.CLIENT.WINDOW.BROWSER" Then           ' Check for correct
        context
        Set Menu = objGWMenu           ' Get Main menu object
        Set Menu = Menu.MenuItems.Item("File")           ' get menu File
        Set Menu = Menu.MenuItems.Item("New")           ' get menu New
        Dim Cmd00 As New GWCommand           ' Build GWCommand object
        Let Cmd00.PersistentID = HelpDesk           ' Set persistent ID for Custom
            menu in GWCommand object
        Let Cmd00.LongPrompt = "This is a help desk"           ' set long prompt for
            menu item
        Call Menu.MenuItems.Add("Hel&pDesk", Cmd00)           ' add menu item to the
            end of menu

        Set Menu = objGWMenu           ' Get Main menu object
        Set Menu = Menu.MenuItems.Item("Tools")           ' get menu Tools
        Set Menu = Menu.MenuItems.AddMenu("S&ample")           ' add menu item to
            the end of the menu

        Set Menu = objGWMenu           ' Get Main menu object
        Set Menu = Menu.MenuItems.Item("Tools")           ' get menu Tools
        Set Menu = Menu.MenuItems.Item("S&ample")           ' get menu S&ample
        Dim Cmd02 As New GWCommand           ' Build GWCommand object
        Let Cmd02.PersistentID = SampleItem           ' Set persistent ID for Custom
            menu in GWCommand object
        Let Cmd02.LongPrompt = "This is a sample menu item"           ' set long
            prompt for menu item
        Call Menu.MenuItems.Add("SampleItem", Cmd02)           ' add menu item to
            the end of menu

    End If

End Function
```

Delphi Example

```
function CommandFactory.CustomizeMenu( Context: string;
                                       GWMenu: variant): ToleBool;

var
  vMenuItems : variant;
  vMenu : variant;
  vSeparator: variant;
  Cmd : Command;
begin

  If CompareStr(Context, 'GW.CLIENT.WINDOW.BROWSER') = 0 then //Check for
correct context
  begin
    vMenu := GWMenu; // get menu form GWMenu
    vMenu := vMenu.MenuItems.Item('File'); //get menu File
    vMenu := vMenu.MenuItems.Item('New'); //get menu New
    GwCmdHelpDesk := Command.Create(HelpDesk); //create command for Custom
                                         menu
    vMenuItems := vMenu.MenuItems.Add('Hel&pDesk',
GwCmdHelpDesk.OleObject); //add menu item to the end of the menu
    GwCmdHelpDesk.LongPrmt := 'This is a help desk'; //set long prompt for
                                         menu item

    GwCmdHelpDesk.Release;

    vMenu := GWMenu; //get menu form GWMenu
    vMenu := vMenu.MenuItems.Item('Tools'); //get menu Tools
    vMenu := vMenu.MenuItems.AddMenu('S&ample'); //add menu item to the end of
                                         the menu

    vMenu := GWMenu; // get menu form GWMenu
    vMenu := vMenu.MenuItems.Item('Tools'); // get menu Tools
    vMenu := vMenu.MenuItems.Item('S&ample'); // get menu S&ample
    GwCmdSampleItem := Command.Create(SampleItem); // create command for
                                         Custom menu

    vMenuItems := vMenu.MenuItems.Add('SampleItem',
GwCmdSampleItem.OleObject); // add menu item to the end of the
                                         menu
    GwCmdSampleItem.LongPrmt := 'This is a sample menu item'; // set long
                                         prompt for menu item
    GwCmdSampleItem.Release;

  end;

  result := FALSE;

end;
```

It is important to note a few things based on the above sample. First, we do not have to define what the menus will actually do here. As discussed above, the `Command.Execute()` method should have entries corresponding to each constant passed when creating the command (e.g. `HelpDesk`, `Sample`, `SampleItem`). Thus, all we need to do here is select which command we want by passing the appropriate constant to the `Command.Create()` method. Second, we can have menu items attached to different `GroupWise` objects that have the same name but that do very different things. The defining factors will be the object (context) and the command attached to the menu item.

Return Value

Your `CustomizeMenu()` method should return a value that is true if you want the method to be called again the next time the menu is updated and false if you only want the method called only once. In almost all cases, you will return a false value so that you will only need to modify the menu once. Even if the menu is rebuilt multiple times, you will have to delete your old menu item and re-install it. You might consider using the `GWCommand.Validate()` method instead.

Practice Tip: Even if you return a false value, you may want to check to make sure that you are not adding a menu twice. Sometimes other programs (or even your own) will call your `C3PO` multiple times, and you don't want to create confusion for the user.

The two ways to check for an existing menu are to walk the toolbar, and to do a find. To walk the toolbar, you will iterate through the menu items, and compare the caption.

VB Example

```
for count = 1 to mainmenuitems.count
    Set contextmenu = mainmenuitems.item(count)
    If contextmenu.caption("&SampleItem" then
        'note how the ampersand is in the caption
        addmenu=false 'this is a boolean you will use to determine whether to add
    End If
next count
```

Delphi Example

```
for count:= 1 to mainmenuitems.count do begin
    contextmenu:=mainmenuitems.item(count);
    if contextmenu.caption='&SampleItem' then begin
        //note how the ampersand is in the caption
        addmenu:=false; //this is a boolean you will use to determine whether to add
        break;
    end;
```

Alternatively, you can attempt to find a menu using the `Item()` method.

VB Example

```
Set vMenu = vMenu.MenuItems.Item("SampleItem") 'Note the lack of ampersand
if vMenu = NULL then
. . .
```

Delphi Example

```
vMenu:= unassigned;
try
vMenu := vMenu.MenuItems.Item('SampleItem');//Note the lack of ampersand
except end;
if varisnull(vMenu) or varisempty(vMenu) then
. . .
```

Finally, you can save the MenuID and find by MenuID.

The wizard will take you through the following steps.



Figure 1: C3PO Creation Wizard.

1. Choose a name for your C3PO. Select the area where the wizard will place the files it creates. Select 'Menus'.

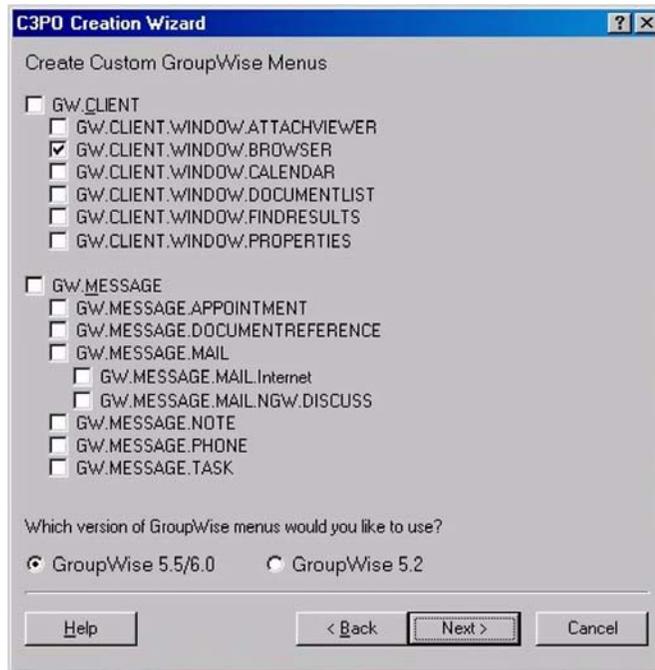


Figure 2: Choose context to modify menus.

2. Choose the context upon which you would like to modify menus. The example above will modify menus on the browser window or messages e.g. mail, appointment, task, etc.

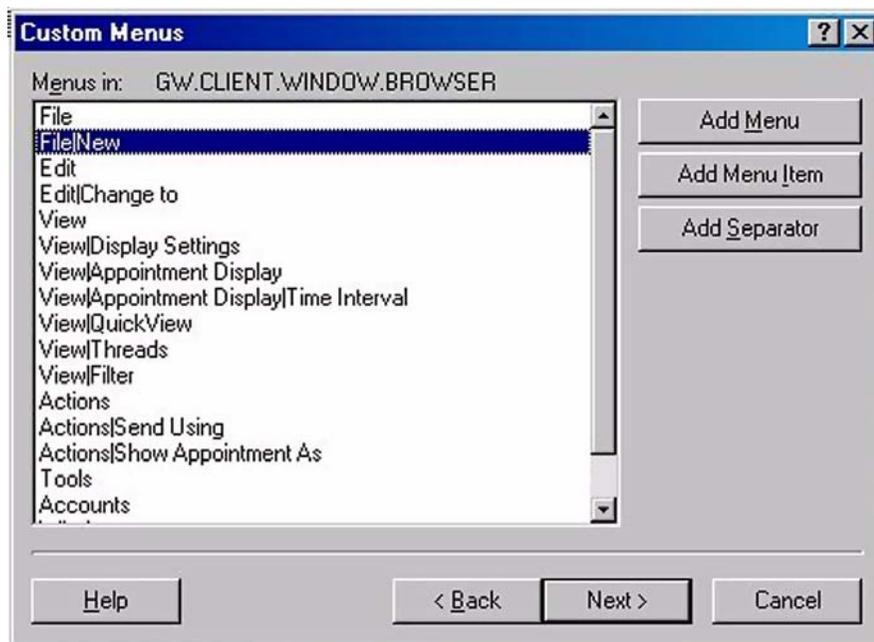


Figure 3: Custom Menus.

- Additional screens will allow you to add menus, menu items, and separators to existing menus based on the context you have chosen; in this case 'GW.CLIENT.WINDOW.BROWSER'. The example above will add a menu item to File | New.

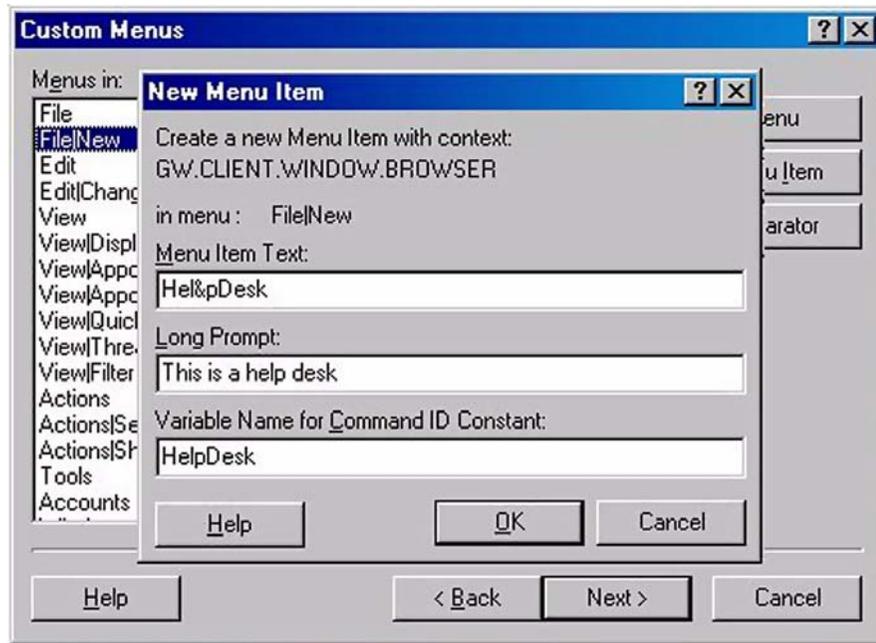


Figure 4: Name, prompt, and ID.

- Choose a name, prompt and ID for the menu item.

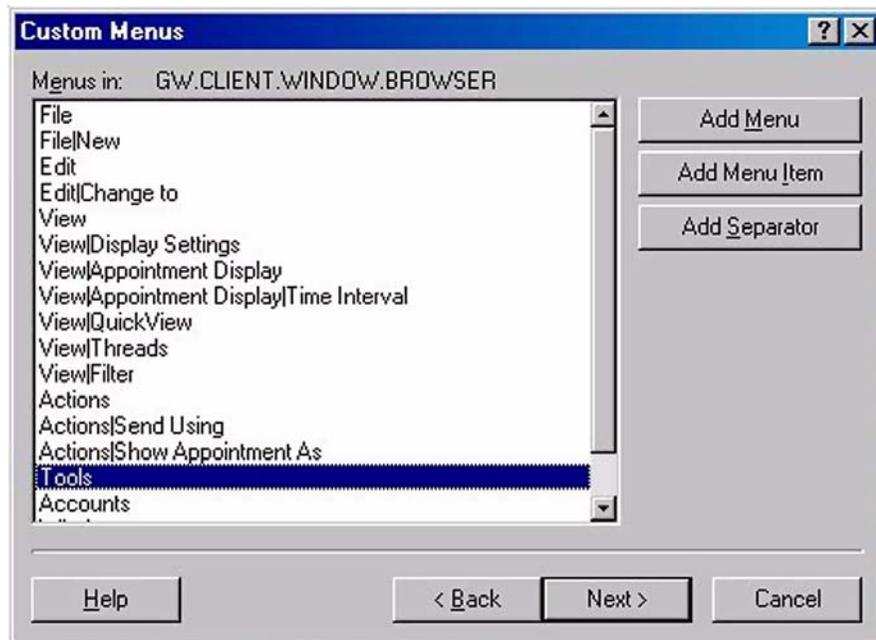


Figure 5: Add menu to tools.

5. The example above will add a menu to Tools.

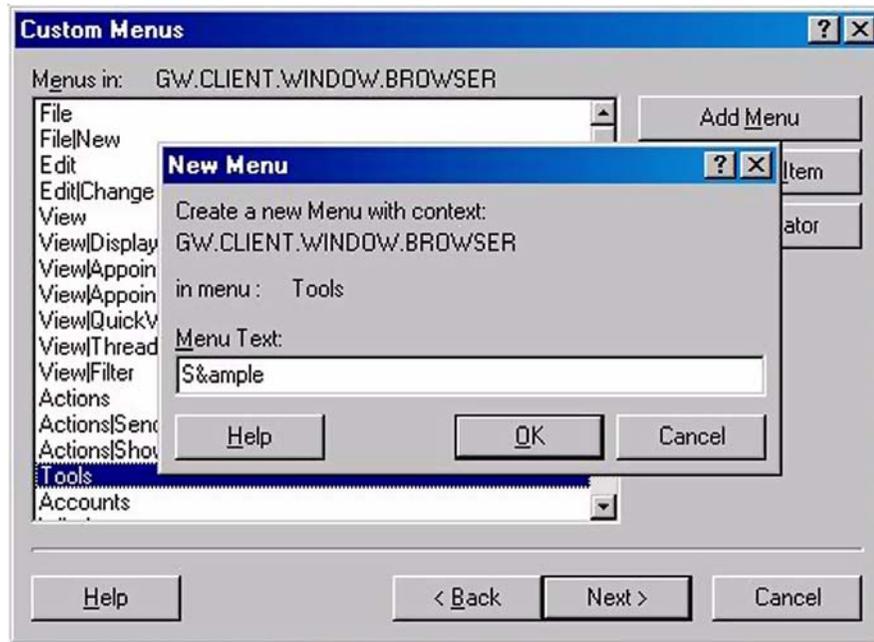


Figure 6: New Menu.

6. Name the menu.

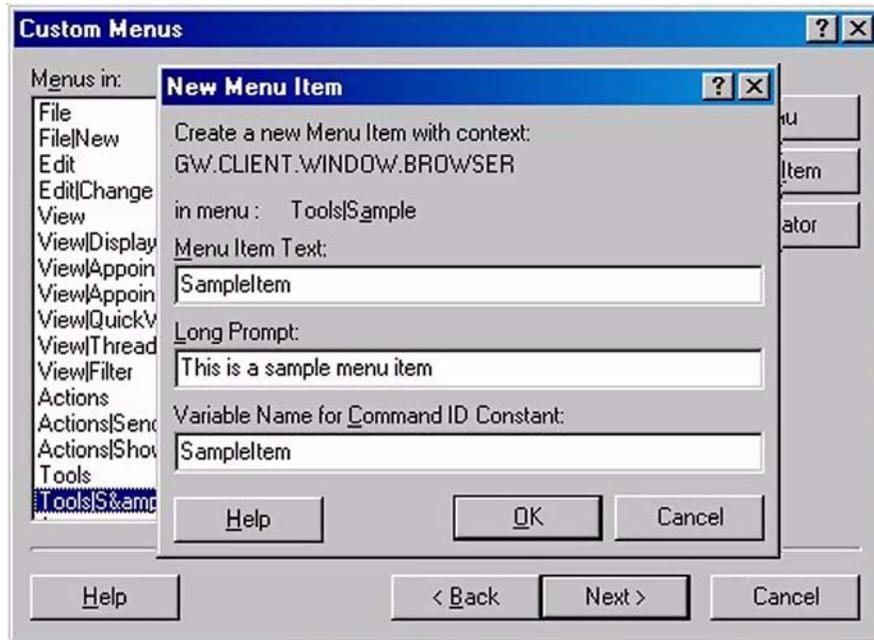


Figure 7: Add menu item to Tools.

7. Add a menu item to Tools | Sample that appears. Give the menu item a name, prompt, and ID.



Figure 8: Finishing C3PO Creation Wizard.

8. When you have finished making selections, you should see something similar to the screen above. Click next and choose language and file output. Register your C3PO and verify that your menus appear in the correct context windows.

Customizing Context Menus

Customizing context menus (the menu you see when you right click on an item or other areas of the GroupWise client) is done in a very similar manner to the way “regular” menus are customized.

Context menus are also made up of `GWMenuItem`s collections, and are manipulated in the same way. You will use the `CustomizeContextMenu()` method to obtain references to the context menu you wish to modify.

```
CustomizeContextMenu(Context ;GWMenu) ;
```

`CustomizeContextMenu()` takes the same parameters as `CustomizeMenu()` method. Check the `Context` to determine whether the object this method receives is one that you want to customize. If it is, then you modify the menu that is passed in the `GWMenu` object.

`CustomizeContextMenu()` has no return value. By definition of a context menu, it is recreated each time the user right-clicks.

The following code adds a “Sample Menu” menu item to the context menu for messages.

VB Example

```
Public Sub CustomizeContextMenu(sGWContext As String, objGWMenu As Object)

    Dim Menu As Object

    If InStr(sGWContext, "GW.MESSAGE") > 0 Then          ' Check for GW.MESSAGE or
                                                         any sub context
        Set Menu = objGWMenu                            ' Get the menu object for the menu at this
                                                         context
        Dim Cmd00 As New GWCommand                    ' Build GWCommand object
        Let Cmd00.PersistentID = MenuItem              ' Set peresistent ID for Custom
                                                         menu in GWCommand object
        Let Cmd00.LongPrompt = "This is another sample item" ' set long
                                                         prompt for menu item
        Call Menu.MenuItems.Add("Sample Item", Cmd00)  ' add menu item to
                                                         the end of the menu

    End If

End Sub
```

Delphi Example

```
procedure CommandFactory.CustomizeContextMenu( Context: string;
GWMMenu: variant);
var
vMenuItems : variant;
vMenu : variant;
vSeparator: variant;
Cmd : Command;
GwCmdMenuItem: Command;           // GWCommand for Custom menu item Sample Item
begin
If Pos('GW.MESSAGE', Context) <> 0 then           // see if the context is
GW.MESSAGE or any sub class
begin
    vMenu := GWMMenu;           // get menu form GWMMenu
    GwCmdMenuItem := Command.Create(MenuItem);
    // create command for Custom menu
    vMenuItems := vMenu.MenuItems.Add('Sample Item',GwCmdMenuItem.OleObject);
    // add menu item to the end of the menu
    GwCmdMenuItem.LongPrmt := 'This is another sample item';
    // set long prompt for menu item
    GwCmdMenuItem.Release;
end;
end;
```

Note how this looks very similar to the menu we just created with `CustomizeMenu()`, except that here there is no need to first grab the “Tools” menu, because we are adding to the top level of the context menu. Note also how we used the exact same command as `CustomizeMenu()` – the `MenuItem` constant. Using the same command ID constant will essentially create the same command (although a different object is created here, they will act the same). Note also that you can save the `GWCommand` object created for your menu in a persistent variable (don’t release it in that method) and thus assign the very same command to the context menu. If you extend the `GWCommand` object, you can thus save the information and use it as long as the object exists.

Practice Tip: If you are customizing a context menu for message contexts, always check and see if you have already created the context menu already. Otherwise, if a user right-clicks on multiple selected messages, you will wind up adding a new context menu multiple times (because the `CustomizeContextMenu()` method will be called once for each method).

In addition to the methods discussed above for `CustomizeMenu()`, Novell has given us a little extra help with context menus. The `CommandFactory` object has a private property called `ContextMenuID`. The Novell samples show this property being set to the `GWMenuItem.MenuID` property. You can use this persistent property to check and see whether or not the context menu has already been set. This only works if you are creating a single context menu action at a time, though, because with multiple menu actions, the `ContextMenuID` will be continually reset. Of course, you can extend the object to include more variables to hold this information if you want.

The wizard will take you through the following steps.



Figure 9: C3PO Creation Wizard.

1. Choose a name for your C3PO. Select the area where the wizard will place the files it creates. Select *Context menus*.



Figure 10: Create Custom GroupWise Context Menus.

2. Select the string that matches the context you want to specify and press Add.

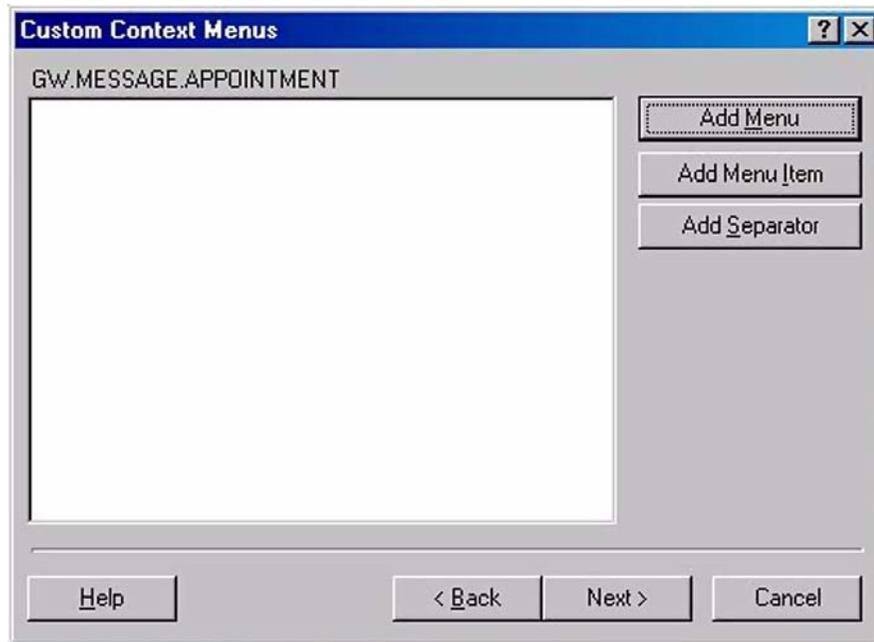


Figure 11: Custom Context Menus.

3. Add menus, menu items or separators that you would like to show up.



Figure 12: New Context Menu.

4. You'll see the screen above if you choose to add a menu. Select a name.

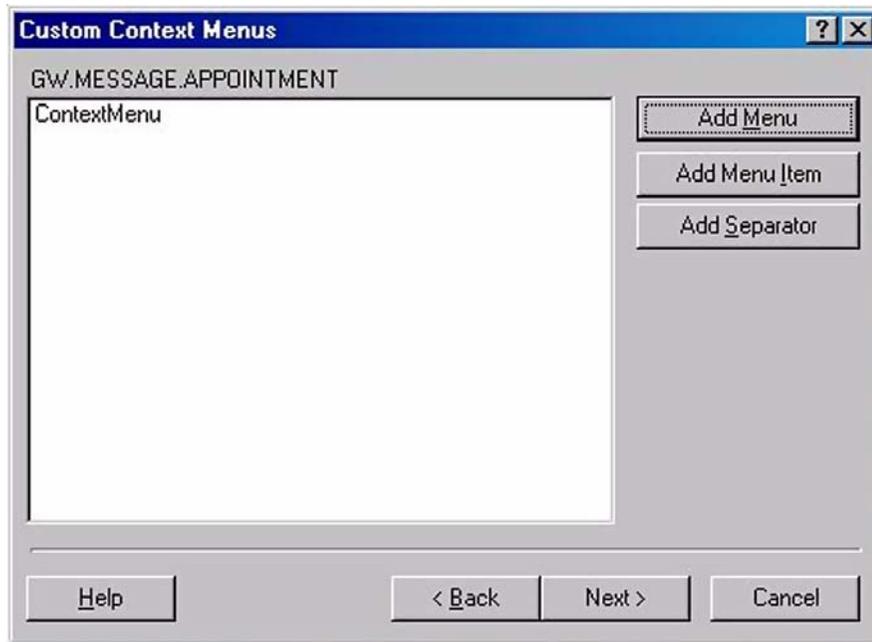


Figure 13:

5. Continue to add menus, menu items, or separators.



Figure 14: *New Context Menu Item.*

6. Give a name to the menu items that you add. Specify an ID for the command that does not contain spaces.

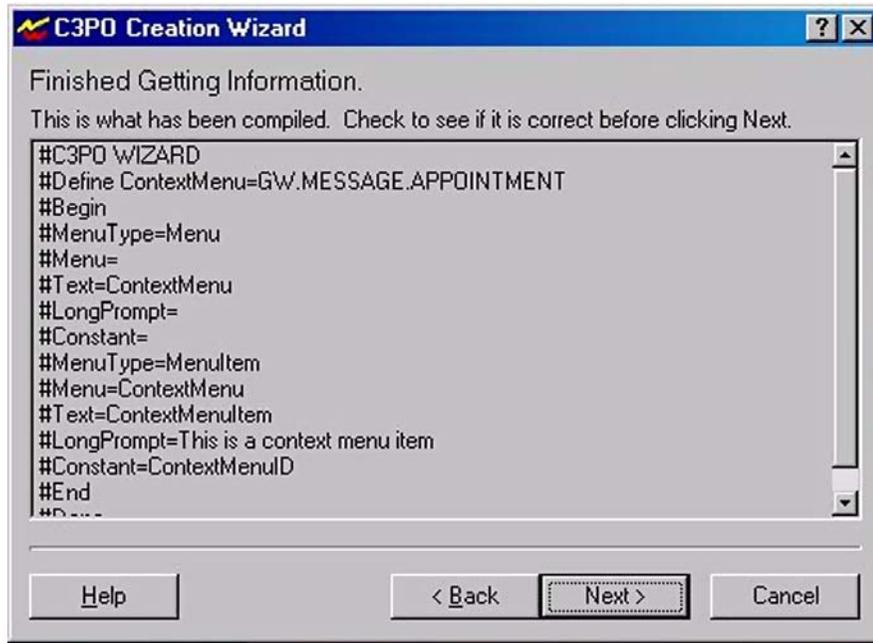


Figure 15: Finishing C3PO Creation Wizard.

- When you have finished making selections, you should see something similar to the screen above. Click next and choose language and file output. Register your C3PO and look for your new context menu when you right click an appointment.

Customizing Toolbars

Customizing toolbars is much like customizing menus, except that you attach commands to toolbar buttons instead of to menu items. Your `CommandFactory.CustomizeToolBar()` method will allow you to customize the toolbar.

```
CustomizeToolBar ( Context , GWToolBar )
```

Name	Data Type	Description
Context	String	This string contains the current GroupWise context. Test this value to determine whether this is a window whose toolbar you want to modify.
GWToolBar	GWToolBar Object	This second parameter is a GWToolBar object. However, you will never modify the GWToolBar directly - rather, you will modify the items in the toolbar. Think of GWToolBar as a top-level menu of toolbar buttons.

The GWToolBar.ToolbarItems Property

Initially, `CustomizeToolBar()` assigns `GWToolBar.ToolbarItems` to a variant variable. This property is of type `GWToolBarItems`, which is a collection of `GWToolBarItem` objects. Like other collection objects, `GWToolBarItems` has an `Item()` method.

```
Item( Index )
```

The `Item()` method takes a single variant parameter `Index`. If `Index` is an ordinal integer (less than `GWToolBarItems.Count`, of course), then the `GWToolBarItem` at that ordinal is returned. If `Index` is a persistent ID, then the `GWToolBarItem` with that persistent ID is returned.

`FindByID()` allows you to find a `GWToolBarItem` with a `ToolBarID` equal to the ID of an existing `GWToolBarItem`.

```
FindByID( ID )
```

This can be an ID for any C3PO's button. Thus, if you know the ID for another C3PO's object, you can obtain the item and modify it in your own code. This function is also helpful for finding out whether a button has already been added.

Adding Buttons

To add to a toolbar, use the `GWToolBarItems.Add()` method.

```
Add( Caption, CommandItem[ , AddBefore ] )
```

This method takes the same parameters as the `GWMenuItem.Add()` method.

Name	Data Type	Description
Caption	String	This is a string that the user sees on the menu.
Item	GWCommand	The second parameter is <code>CommandItem</code> , a <code>GWCommand</code> object that has been created before calling the <code>Add()</code> method. <code>GWCommand</code> is discussed above.
AddBefore	Variant	The third parameter is <code>AddBefore</code> , an optional variant. If <code>AddBefore</code> is an integer, then the button will be added before that item. The integer must be less than <code>GWToolBarItems.Count</code> . If <code>AddBefore</code> is a string, then the new button will be added before the button with that display name - you should ignore the ampersand (&). If you omit <code>AddBefore</code> , the button will be added to the end of the current toolbar.

`Add()` returns the new `GWToolBarItem` object, which you should assign to a variable to manipulate.

Activating the Button. Unlike menu action items, you must “activate” your toolbar buttons with an additional method. This method is called `GWToolBarItem.SetBitmap()`.

`SetBitmap(FileName,ResID)`

Name	Data Type	Description
Filename	String	Filename is a string containing the path to the .dll file that holds the icon. parameter This must be a full path name unless the file is in the GroupWise directory.
ResID	Variant	ResID is the resource index of the bitmap you want to use on the button. If the bitmap is represented by a string in your .dll file, then ResID must be a string corresponding to that name. If the bitmap is represented by a number, then ResID must be an integer corresponding to that index value. Typically, you should use integer index values, because you can perform more functions with the bitmap, as discussed below.

The following code puts this all together for you.

VB Example

```
Public Function CustomizeToolbar(sGWContext As String, objGWToolBar As Object)
As Boolean
    Dim Button As Object
    Dim FilePath As String

    If sGWContext = "GW.CLIENT.WINDOW.BROWSER" Then           ' Check for correct
context
        Dim Cmd00 As New GWCommand                          ' Build GWCommand object
        Let Cmd00.PersistentID = tool1                       ' Set persistent ID for GWCommand
object
        Let Cmd00.ToolTip = "This is the browser sample"    ' Set Button
tooltip
        Set Button = objGWToolBar.ToolbarItems.Add("Sample 1", Cmd00) '
Add button to toolbar
        FilePath = App.Path & "\icons.dll"                 ' Set bitmap for Button
'C3PO WIZARD icons.dll can be replaced by the full path name of any .exe or .dll
that contains a 16x16 and a 32x32 pixel bitmap.
        ' BUTTON_1 can be replaced with the name of the bitmap contained in the
.exe or .dll
        Call Button.SetBitmap(FilePath, "BUTTON_1")         ' set were the bitmap
is found and its name

    End If
    If InStr(sGWContext, "GW.MESSAGE") > 0 Then           ' Check for GW.MESSAGE or
any sub context
        Dim Cmd10 As New GWCommand                          ' Build GWCommand object
        Let Cmd10.PersistentID = tool2                       ' Set persistent ID for GWCommand
object
        Let Cmd10.ToolTip = "This is the mail sample"      ' Set Button tooltip
        Set Button = objGWToolBar.ToolbarItems.Add("Sample 2", Cmd10) '
Add button to toolbar
        FilePath = App.Path & "\icons.dll"                 ' Set bitmap for Button
```

```

'C3PO WIZARD icons.dll can be replaced by the full path name of any .exe or .dll
that contains a 16x16 and a 32x32 pixel bitmap.
  ' BUTTON_1 can be replaced with the name of the bitmap contained in the
.exe or .dll
  Call Button.SetBitmap(FilePath, "BUTTON_1")           ' set were the bitmap
is found and its name

  End If

  CustomizeToolbar = False

End Function

```

Delphi Example

```

function CommandFactory.CustomizeToolBar( Context: string;
                                          GWToolbar: variant): ToleBool;
var
  Button: variant;
  ToolbarItems: variant;
  Cmd: Command;
begin
  ToolbarItems := GWToolbar.ToolbarItems;    // get toolbar items
  If CompareStr(Context, 'GW.CLIENT.WINDOW.BROWSER') = 0 then
    // Check for correct context
  begin
    GwCmdtool1 := Command.Create(tool1);    // create command for toolbar
    Button := ToolbarItems.Add('Sample 1', GwCmdtool1.OleObject);
    // add new Button
    GwCmdtool1.ToolTp := 'This is the browser sample';
    // set tooltip for toolbar item
    Button.SetBitmap(GetCurrentDir+'\SampleIcons.dll',1); // set toolbar
    bitmap
    GwCmdtool1.Release;
  end;
  If Pos('GW.MESSAGE', Context) <> 0 then
    // see if the context is GW.MESSAGE or any sub class
  begin
    GwCmdTool2 := Command.Create(tool2);
    // create command for toolbar
    Button := ToolbarItems.Add('Sample 2', GwCmdTool2.OleObject);
    // add new Button
    GwCmdTool2.ToolTp := 'This is the mail sample';
    // set tooltip for toolbar item
    Button.SetBitmap(GetCurrentDir+'\SampleIcons.dll',2); // set toolbar
    bitmap
    GwCmdTool2.Release;
  end;
  result := FALSE;
end;

```

More about Icons. Buttons pictures must actually be bitmaps, and not icons. The bitmaps may be 16 x 18 or 16x 16. Each button picture has its own bitmap.

The C3PO wizard comes with an icons.dll file that you can use for testing.

The following information is specific to Delphi which has problems with some icon files, and thus is very sensitive. Here is some Delphi code for creating icons – this assumes that you have a .bmp file or two with the icons you want. You can use the image editor in Delphi or any other suitable tool you want.

Create a new “SampleIcons.dpr” project which is a .dll. The file should look like this:

```

library sampleicons;
  {$R sampleic.res}
  uses
    SysUtils,
    Classes;

begin
end.
```

That’s it! That is the whole .dll file. So long as you have bitmaps in sampleicons.res, you will get a .dll with your icons.

Buttons can also have bitmap masks for special display features. In order to use this function, your .dll must refer to the toolbar bitmaps as numbers. That is, they must be named ‘1’, ‘2’, etc. If you use string names, this feature will not work. Further, your bitmap must be defined as 16 color for these features to work. We think this is an important feature, and thus our sample is coded this way. The RGB values for each feature are:

RGB(0, 255, 0)	Transparent. This is the most important mapping. Any color in the button bitmap will be “see-through” in GroupWise, and will thus take any color that the button bar is set to in Windows (though it is usually grey). In a color editor, this color looks greenish
RGB(192, 192, 192)	COLOR_3DLIGHT. This feature will make the “lit” look that depressed buttons have. Note that you do not need to worry about this usually, as Windows will change the colors as necessary. This is a light gray color in appearance at design time
RGB(255, 255, 255)	COLOR_3DHILITE. This feature will make the “whitish” color that you see to the bottom right of a depressed button. This is a white color at design time
RGB(128, 128, 128)	COLOR_3DSHADOW. This feature will make the darker type of shadow you see at the top left of a button. Like COLOR_3DHILITE, Windows will take care of this automatically for depressed buttons. This is a dark gray color in appearance at design time

The sample bitmap with this book show what the colors corresponding to each bitmap actually look like during the design phase.

Deleting Buttons

To delete a toolbar button (including some native to GroupWise), call the buttons `GWToolbarItem.Delete()` method. Not all system buttons can be deleted. You will have to experiment a bit to determine what can be removed and what can not.

Power Tip: ToolbarID

You can manipulate button items using the WindowsAPI's by accessing the `GWToolbarItem.ToolbarID` property, which is a long integer. This property is assigned when the button is created, and it corresponds to the unique ID that Windows assigns to each toolbar button. This is extraordinarily useful if you would like to do things with the toolbar that you could not do otherwise.

The first thing you need to do is extend the `GWCommand` object to include a `ButtonID` property, which is an integer. You can do this by subclassing the `GWCommand` or by just adding the property in your current C3PO declaration. This property can be private. Next, assign the `GWToolbarItem.ToolbarID` to `GWCommand.ButtonID`. Thus, your `CustomizeToolbar()` method will include the following:

```
Cmd := Command.Create(MyCommandConstant)
MyToolbar := ToolbarItems.Add('Sample ', Cmd.OleObject);
Cmd.ButtonID:=MyToolbar.toolbarid;
```

Now, whenever `Command.Validate` or `Command.Execute` is called, you can access the `Command.ButtonID` property and go from there. The following is some sample WindowsAPI code that will allow you to grab the actual button:

```
var
handle:thandle;
begin
enumchildwindows(win,@enumproc,longint(@handle));
end;
```

Your forward declaration will look like this:

```
function enumproc(childhandle:thandle;address:longint):boolean; stdcall;
```

And your actual callback declaration will look like this:

Delphi Example

```
function enumproc(childhandle:thandle;address:longint):boolean; stdcall;
    //callback to handle enumchildwindows and grab the toolbar
type pint=^longint;
var classname:pchar;
begin
classname:=stralloc(256);
getclassname(childhandle,classname,256); // look at this window class
if classname=toolbarclassname then begin //if this is it, then OK.
    //toolbarclassname is defined in commctrl, which you must include
result:=false; //false here means stop
pint(address)^:=childhandle; //return the handle
end else begin
```

```
result:=true; //true here means keep going
end;
strdispose(classname);
end;
```

Now handle from your original `EnumChildWindows()` call will contain a handle to the toolbar. From there, you can use the toolbarid saved in `GWCommand.ButtonID` to your benefit. Here is an example that “presses” the button and keeps it depressed. This is useful as a state button (like the bold font button). Note that you must use the numbering method for your bitmap names in order for shading to work correctly in this implementation.

```
enumchildwindows(win,@enumproc,longint(@handle));
sendmessage(handle,TB_SetState,Command.ButtonID,longint(TBSTATE_PRESSED))
//TB_SetState and TBSTATE_ENABLED and TBSTATE_PRESSED are in commctrl
```

The wizard will take you through the following steps.

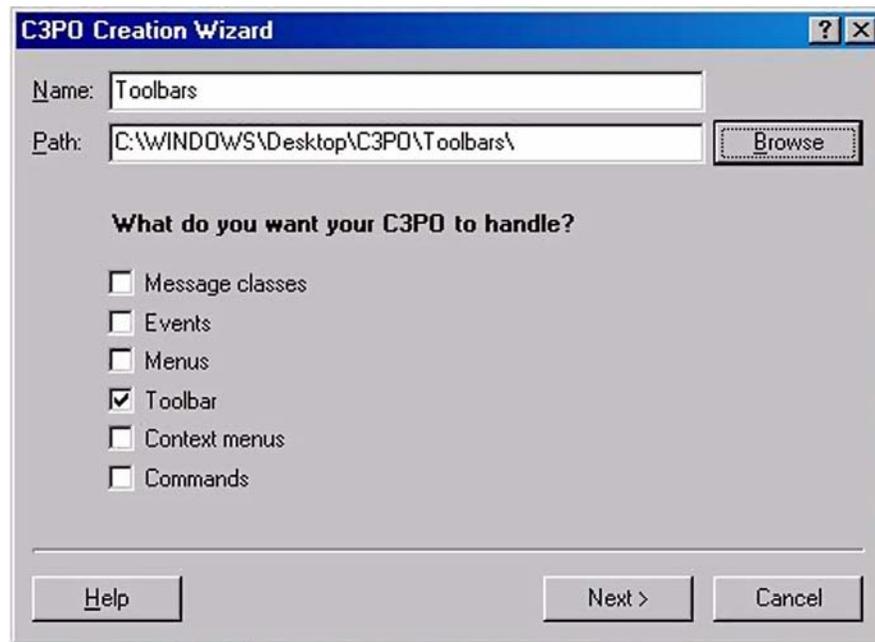


Figure 16: C3PO Creation Wizard.

1. Choose a name for your C3PO. Select the area where the wizard will place the files it creates. Select *Toolbars*.

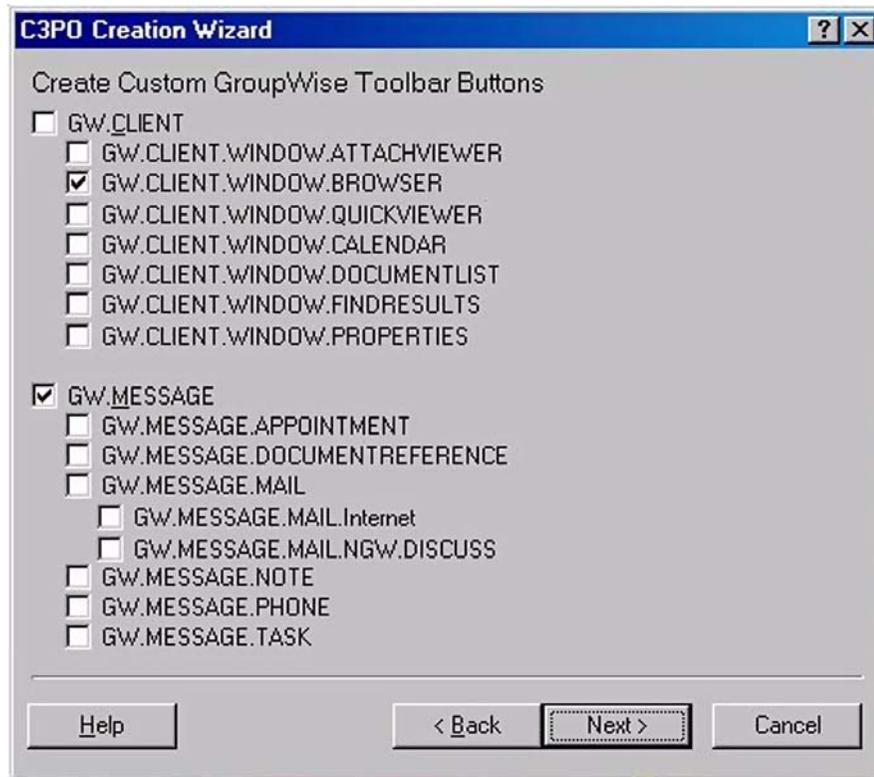


Figure 17: Context to modify toolbars.

2. Choose the context upon which you would like to modify toolbars. The example above will modify toolbars on the browser window and mail messages.

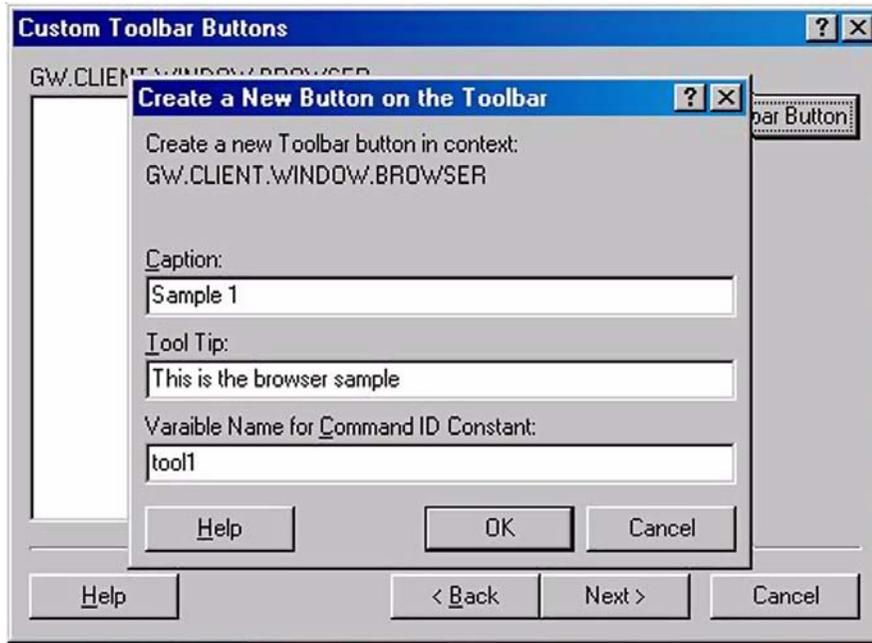


Figure 18: Create a New Button on the Toolbar.

3. Choose *Add Toolbar Button* and give a caption, tool tip and ID to the button that goes on 'GW.CLIENT.WINDOW.BROWSER'

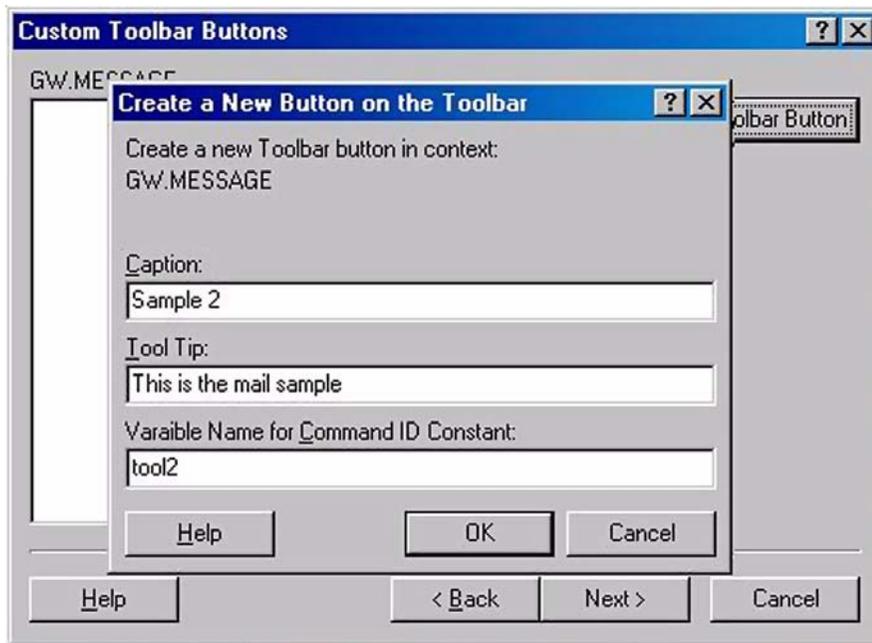


Figure 19: Create a New Button on the Toolbar.

4. Choose *Add Toolbar Button* and give a caption, tool tip and ID to the button that goes on 'GW.MESSAGE'.

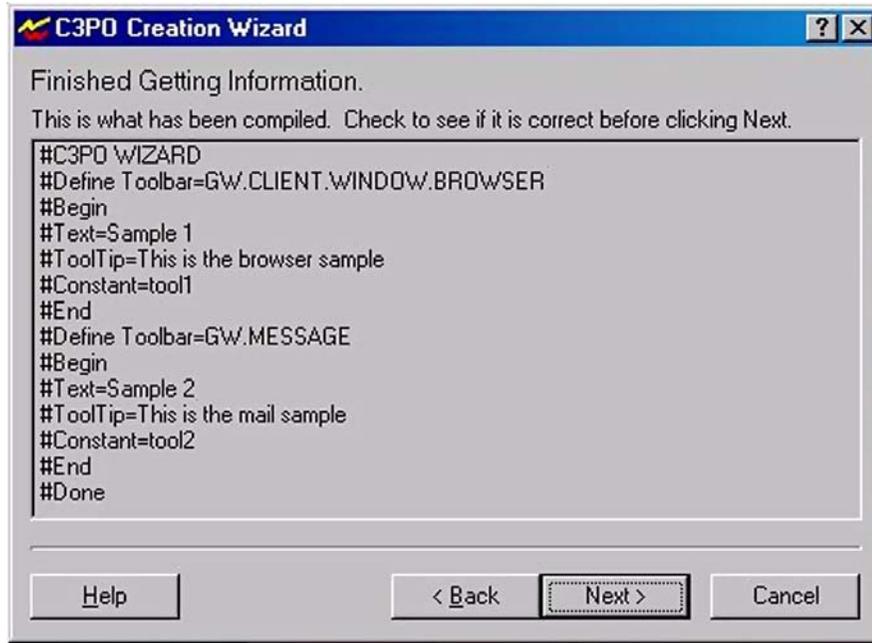


Figure 20: Finishing the C3PO Creation Wizard.

5. When you have finished making selections, you should see something similar to the screen above. Click *Next* and choose language and file output. Register your C3PO and verify that the correct buttons appear on the correct contexts.

Summary

You now have seen how to add menus, context menus, and toolbar buttons to GroupWise. With the wizard you are able to focus on adding functionality to your C3PO while you let the tool build the class files for you. In the next chapter, we will look at capturing commands, events and creating custom message classes.

Copyright © 2002 by Novell, Inc. All rights reserved.
No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, for any purpose without the express written permission of Novell.

All product names mentioned are trademarks of their respective companies or distributors.

Capturing a Predefined Command

Chapter 18

Section 3: GroupWise C3POs

Capture the following predefined commands.

Event	Description
eGW_CMDID_ACCEPT	When an item is accepted
eGW_CMDID_ARCHIVE	When an item is archived
eGW_CMDID_COMPLETE	When an item is completed
eGW_CMDID_COMPOSE	When an item is created
eGW_CMDID_DECLINE	When an item is declined
eGW_CMDID_DELEGATE	When an item is delegated
eGW_CMDID_DELETE	When an item is deleted
eGW_CMDID_DOC_CHECKIN	Display the check-in time of a document
eGW_CMDID_DOC_CHECKOUT	Display the check-out time of a document
eGW_CMDID_DOC_RESETEINUSE	Display when attempting to reset a document's in-use flag
eGW_CMDID_FORWARD	When an item is forwarded
eGW_CMDID_OPEN	When an item is opened
eGW_CMDID_OPENFILEATTACH	When an item attachment is opened
eGW_CMDID_PRINT	When an item is printed
eGW_CMDID_PROPERTIES	Display the properties for a GroupWise object
eGW_CMDID_REPLY	When an item is replied
eGW_CMDID_RESEND	Resend an instance of the item type that is currently in the GroupWise Out Box
eGW_CMDID_SAVE	When an item is saved back to the database
eGW_CMDID_SAVEAS	When an item is saved as an external file
eGW_CMDID_SAVEATTACHAS	Save an instance of the item type back to the data base (attachment)
eGW_CMDID_SEND	When an item is sent
eGW_CMDID_SETALARMS	Set alarms on a GroupWise object
eGW_CMDID_UNDELETE	When an item is removed from the trash
eGW_CMDID_VIEW	When an item is viewed
eGW_CMDID_VIEWATTACH	When an item attachment is viewed

GroupWise calls `WantCommand()` with a context and ID to determine which commands your C3PO wants to handle. If a match is made and `TRUE` is returned, `BuildCommand()` will create a `GWCommand` object and set the base command for later use. Within `GWCommand`, `Execute()` tries to find a match for the command your C3PO is planning to intercept. `Execute` is where your custom code should be placed.

The wizard takes you through the following screens.

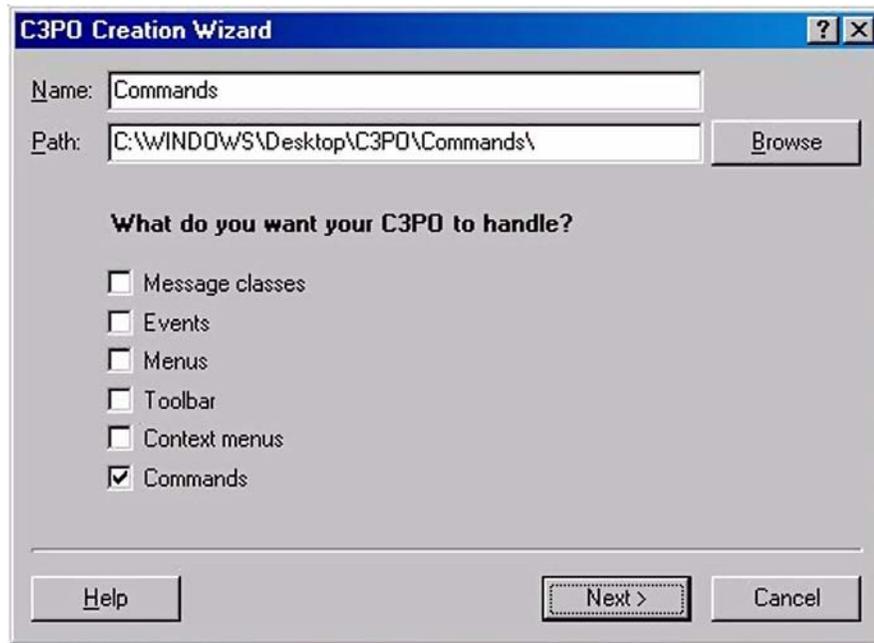


Figure 1: C3PO Creating Wizard.

1. Choose a name for your C3PO. Select the area where the wizard will place the files it creates. Select *Commands*.



Figure 2: Content to capture commands.

2. Select the context that you're interested in using to capture commands. Press *Add*.

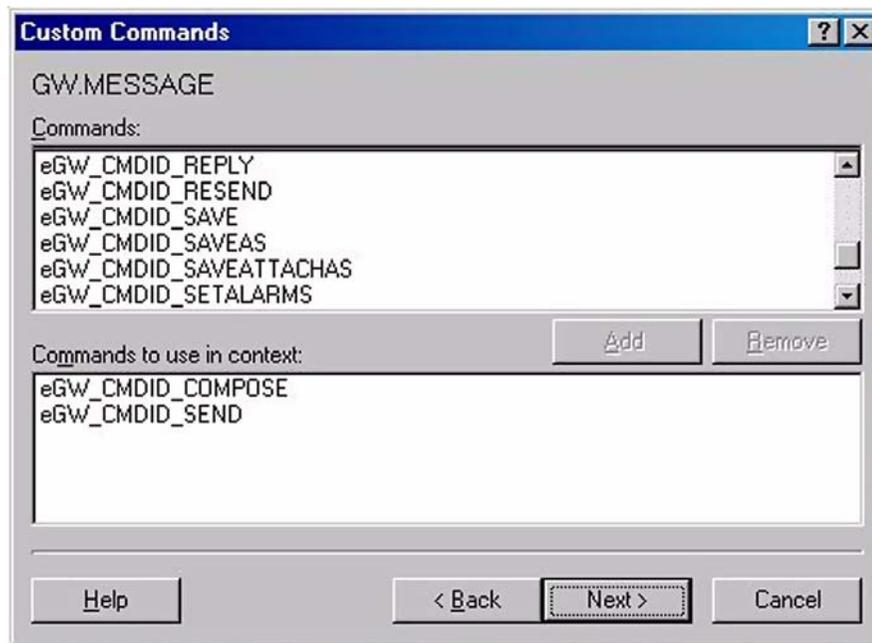


Figure 3: Custom commands.

3. Choose the commands that you want to intercept. The example above will work with 'compose', when you create a message and 'send' when you send a message.

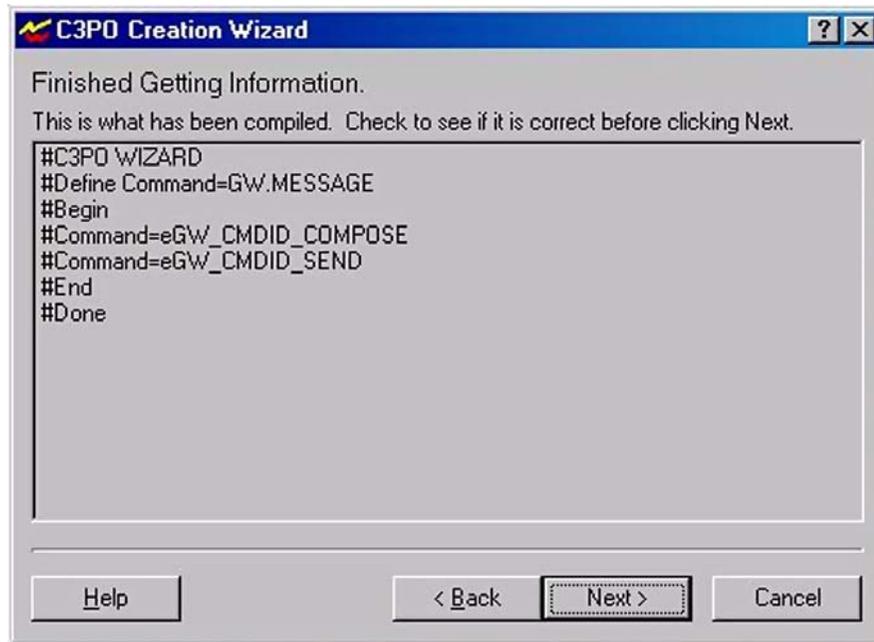


Figure 4: Finishing C3PO Creation Wizard.

4. When you have finished making selections, you should see something similar to the screen above. Click next and choose language and file output. Register your C3PO.

Creating a Custom Message

There are 4 major steps to creating a custom message in GroupWise:

- Pick a parent class to subtype and a subclass name.
- Create a new C3POServer class or modify an existing one.
- Register the new object.
- Create or modify existing objects to handle the new classes appearance, handling of commands, or response to events.

Pick a Parent Class to Subtype

New classes inherit the characteristic properties of the classes that they are derived from. The following six message classes are available for you to subtype:

- GW.MESSAGE.APPOINTMENT
- GW.MESSAGE.DOCUMENTREFERENCE
- GW.MESSAGE.MAIL

- GW.MESSAGE.NOTE
- GW.MESSAGE.PHONE
- GW.MESSAGE.TASK

Many C3PO's subclass GW.MESSAGE.MAIL. A subclass of MAIL will give you all the properties and methods normally associated with this GroupWise object and its parent class MESSAGE. To get at other message characteristics such as the appearance on the calendar with a StartDate and a Duration, it would be better to subtype another class of MESSAGE. In this case you would choose to subtype GW.MESSAGE.APPOINTMENT which provides the added functionality associated with an appointment. The Subclass name should be descriptive of what the C3PO is or does in order to make code more readable. We use "MyC3PO" a lot in instructional materials, but a name like "HelpDesk" is much more descriptive.

The resulting class name (also called context) that we are going to use in our project below is:

GW.MESSAGE.TASK.HELPDESK

Note: If a message of type GW.MESSAGE.TASK.HELPDESK arrives in the mailbox of a client where the C3PO to handle it is not installed and registered, then the client will assume it is a message of type GW.MESSAGE.TASK and handle it accordingly. If you used a custom view, that will open as well.

Create a New C3POServer Class

The next step is to either create a new project or modify an existing project with a C3POServer class. The easiest way to do this is with the C3PO wizard.

The project must be either an ActiveX EXE or ActiveX DLL type. The project name will determine the CLSID the C3PO is registered under (As well as the EXE or DLL name). Select a descriptive project name.

Note: The CLSID may be used in code to implement early binding. Choose a project name without any spaces to make this easier.

Modifying the Appearance of your Custom Message

The appearance of the message can be modified in several ways.

- A custom icon may appear next to the message in the message list of the browser. Separate icons may be used for unopened or opened items.
- The toolbars, menus, and context menu of the item can be modified in the same manner they are modified in the browser.

- Alternatively, the Compose or Open commands can be trapped, and a custom designed form opened in place of the normal view. Controls can be placed on the form to provide a customized interface.

Custom icons are made available by creating an IconFactory object with a single method: GetIcons(). The syntax is

```
GetIcons (Context , IconFile , UnOpenedIconIndex , OpenedIconIndex )
```

Parameters	Data Type	Description
Context	String (input)	The context (message type) of the message to return the icon for.
IconFile	String (output)	The full or relative path and file name of the .exe or .dll file with 16x16 and 32x32 bit icon resources.
UnOpenedIconIndex	Long (output)	The index to the icon for an unopened item in the IconFile resource.
OpenedIconIndex	Long (output)	The index to the icon for an opened item in the IconFile resource.

Toolbars, menus, and context menu of the item are modified for a custom message in the same manner as for the browser.

If GW#C#Open and/or GW#C#Compose commands are registered for handling by the C3PO, and properly coded in the CommandFactory.WantCommand () and CommandFactory.BuildCommand() methods, then when the Command.Execute() method is called, custom code could produce a custom form. From a users perspective, the enter action will be seamless.

In the case of the Compose command, the form must provide the functionality to fill in fields, and save or send the message using the Token or GroupWise Object API. Additional functionality may include querying an application or database for information and adding custom fields to the message.

In the case of the Open command, the functionality of the form is to read and display the parts and custom fields of the message, and possibly take care of responding to the message with a new message or interacting with another application or database.

The wizard will take you through the following screens.



Figure 5: C3PO Creation Wizard.

5. Choose a name for your C3PO. Select the area where the wizard will place the files it creates. Select *Message classes*.

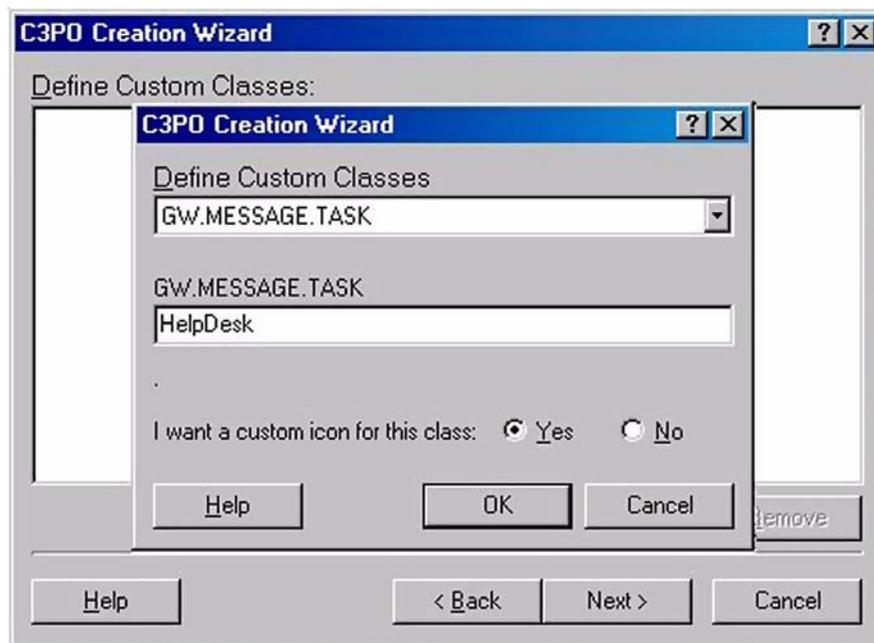


Figure 6: Choose message context.

6. Select *Add* and choose the message context that you'd like to subclass. Select *Yes* or *No* depending on whether or not you want a custom icon associated with the new message context you're creating.

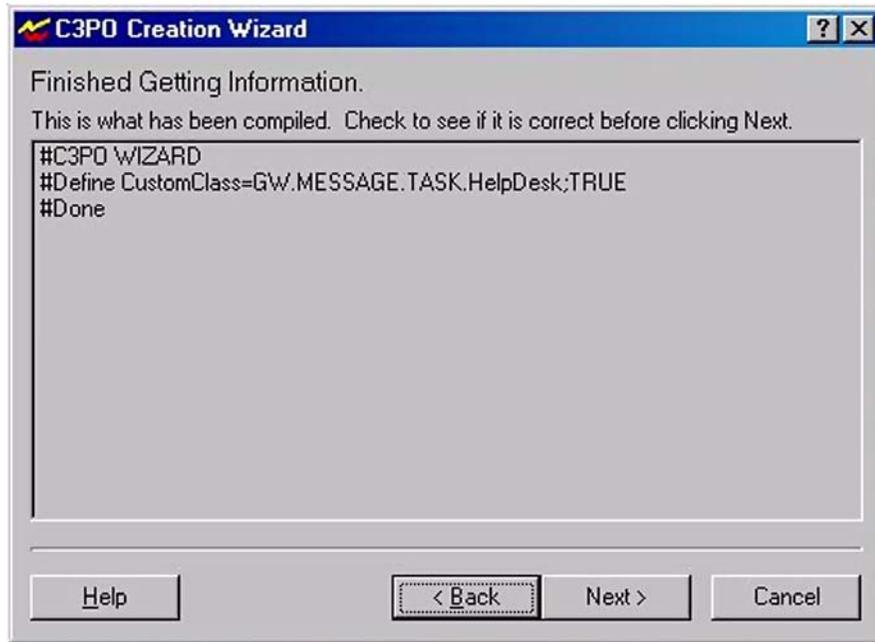


Figure 7: Finishing C3PO Creation Wizard.

- When you have finished making selections, you should see something similar to the screen above. Click *Next* and choose language and file output. Register your C3PO.

GroupWise Events

GroupWise events are trapped by Event.Notify in EventMonitor. When GroupWise calls Event.Notify, it passes in two parameters.

```
EventMonitor.Notify(Context, Evt)
```

Name	Data Type	Description
Context	String	This is the context which is generating the event, such as the browser, message, etc.
Evt	GWEvent object	This is the actual event that has occurred. It contains a single property: Evt.PersistentID

Different events are passed in as the values of Evt.PersistentID:

Name	Data Value	Description
eGW_CMDEVTID_READY	'GW#E#0'	This event is called once, after the GroupWise client has started and all services (DDE and OLE) have been activated.

Name	Data Value	Description
eGW_CMDEVTID_DELIVERY	'GW#E#1'	You will receive this event when a new item is delivered to the GroupWise mailbox. This event will not be triggered for personal items. From there you can obtain a reference to ClientState.CommandMessage to review the message properties. In addition, you may call the ClientState.SetupDeliveryFolder() method to deliver the message to another folder.
eGW_CMDEVTID_SHUTDOWN	'GW#E#2'	This event is called once, just before the GroupWise client exits. However, when this event is sent, all GroupWise access is still available for last minute processing.
EGW_EVT_OVERFLOW		Occurs when more messages of interest are delivered to the In Box than the Delivery event can handle.

The following is a sample method that implements these events. A skeleton for two events OnShutdown when GroupWise is ready to shutdown and onDelivery when GroupWise has received a new message item, show you how to trap a new "HelpDesk" message type, so you can process it specially.

VB Example

```
Public Sub Notify(sGWContext As String, objGWEEvent As Object)
    Dim res

    Select Case objGWEEvent.PersistentID
        Case eGW_CMDEVTID_SHUTDOWN          'Check for Shutdown Event
            'C3PO WIZARD This is were you put your Shutdown code.
            res = MsgBox(objGWEEvent.PersistentID, vbOKOnly, sGWContext)
        Case eGW_CMDEVTID_DELIVERY          'Check for Delivery Event
            'C3PO WIZARD This is were you put your Delivery code.
            If sGWContext = "GW.MESSAGE.TASK.HelpDesk" Then          ' Check for
correct context
                'C3PO WIZARD This is were you put your Delivery code for
                GW.MESSAGE.TASK
                res = MsgBox(objGWEEvent.PersistentID, vbOKOnly, sGWContext)
            End If
        Case Else
            MsgBox "Unsupported Case"
    End Select

End Sub
```

Delphi Example

```
procedure EventMonitor.Notify(Context:string; evt:variant);
begin
    if CompareText(evt.PersistentID,eGW_CMDEVTID_SHUTDOWN) = 0 then// Is this
OnShutdown notify
        begin
            //Add code here for the shutdown event
        end
end
```

```
else if CompareText(evt.PersistentID,eGW_CMDEVTID_DELIVERY) = 0 then
// Is this OnDelivery notify
begin //we have already added the code for delivery event
  if CompareText(Context,'GW.MESSAGE.TASK.HelpDesk') = 0 then
    // Check if this is the correct message class
    begin
      //Add code here to process helpdesk messages
    end;
  end;
end;
end;
```

The wizard will take you through the following screens.



Figure 8: C3PO Creation Wizard.

1. Choose a name for your C3PO. Select the area where the wizard will place the files it creates. Select *Events*.

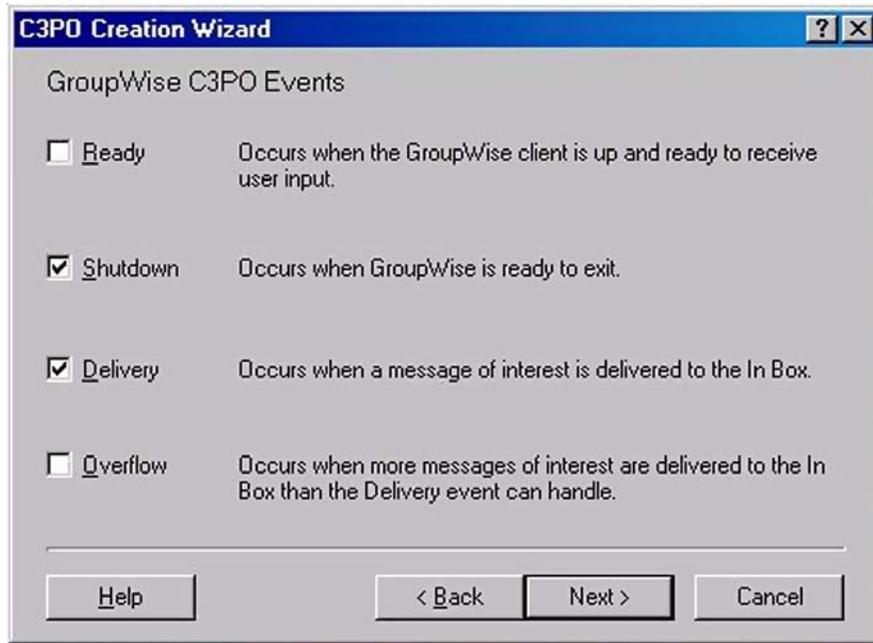


Figure 9: Choose GroupWise C3PO events.

2. Choose events that you want to capture. The example above will add code to Notify to capture onShutdown and onDelivery.

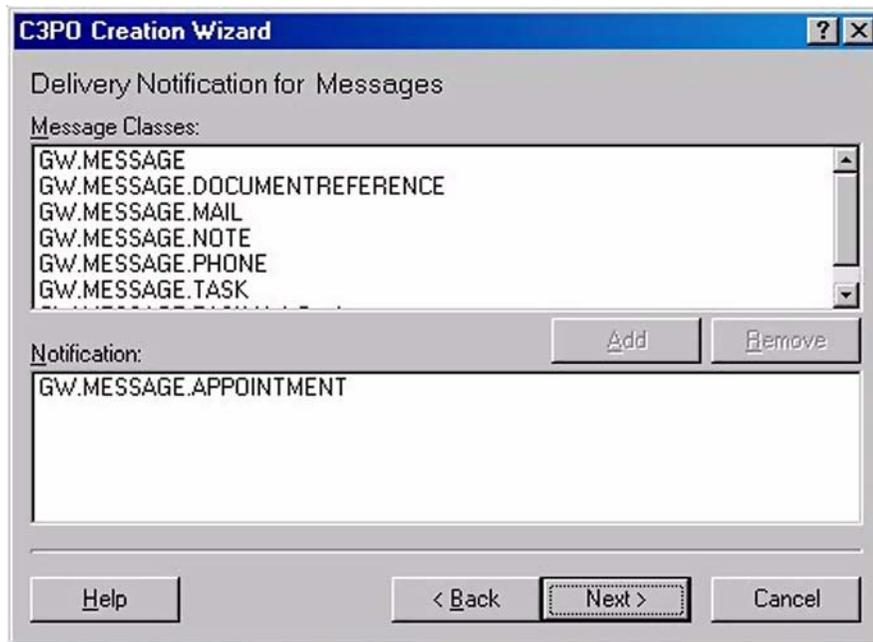


Figure 10: Select context for certain events.

3. Additional screens will allow you to select context for certain events for example, choose 'GW.MESSAGE.APPOINTMENT' to catch the delivery of GroupWise appointments.

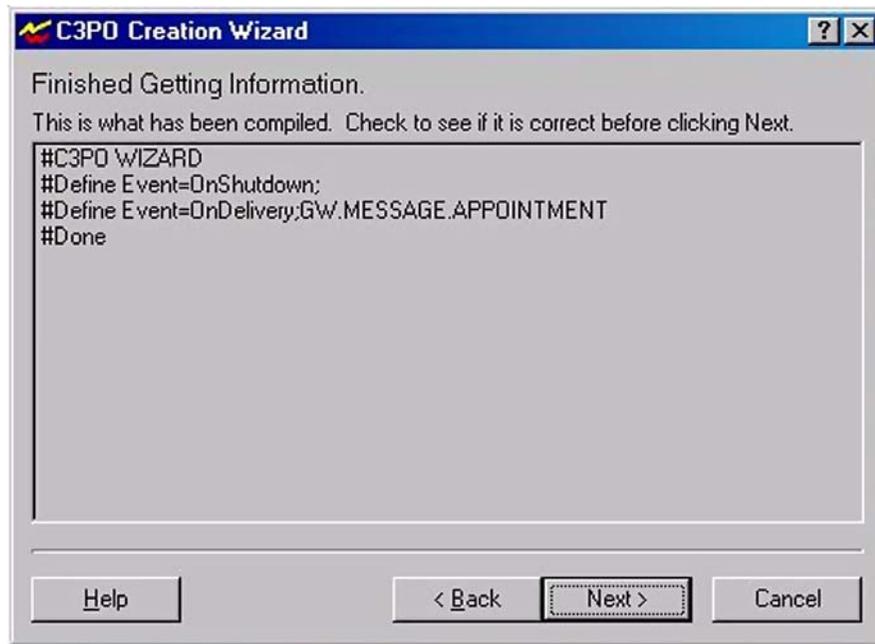


Figure 11: Finishing C3PO Creation Wizard.

4. When you have finished making selections, you should see something similar to the screen above. Click *Next* and choose language and file output. Register your C3PO and verify that the correct events are getting captured.

Summary

As you have seen, creating a custom message in GroupWise is rather easy. Not only do you know how to create a your own context, you also know how to add menus and toolbar buttons to the message that you subtype. In addition you now know how to capture and process GroupWise events.

Copyright © 2002 by Novell, Inc. All rights reserved.
No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, for any purpose without the express written permission of Novell.

All product names mentioned are trademarks of their respective companies or distributors.

Putting It All Together

Chapter 19

Section 3: GroupWise C3POs

Now that we have learned the basics, let's take a look at putting all of the tools together to get some real work done. The real power of the GroupWise APIs are that you can combine more than one API to achieve your needs. In fact, sometimes you might have to combine APIs to get the job done because one API may not have what you need while another will.

Another Look At Validate

The `GWCommand.Validate` method in C3POs can be exploited if you combine the OAPI and token interfaces with your C3PO. Let's take a look at how to enhance validation.

New Message Menus

Let's say you want to enable a menu only on new messages. Unfortunately, C3PO contexts only tell you the view type, but they fail to tell you whether you are working with a new message or a message in the database. We turn to tokens to help us out.

The best way to tell if you have a new message is to check its MessageID. If the message has a MessageID of "X00" then you know it is a new message.

Delphi Example

```
function Command.Validate: longint;  
var messageid:string;  
begin  
if m_nCmd=MYCOMMANDID then begin  
Commander.Execute('itemmessageidfromview()',messageid);  
if messageid="X00" then  
result := eGW_CMDVAL_ALWAYS;  
end;  
end;
```

That's all there is to it. Check the MessageID, and enable the menu if it is a new message.

More About the Button Bar

The validation of the button bar we discussed above is a little more complicated than originally presented. This is because the `Validate` method runs continuously on the button bar and continually calls the `ItemMessageIdFromView()` token and continuously tests for a depressed button would take significant processing cycles. As a refresher, here is the `Validate` method for depressing the button:

```
enumchildwindows(win,@enumproc,longint(@handle));
sendmessage(handle,TB_SetState,Command.ButtonID,longint(TBSTATE_PRESSED))
//TB_SetState and TBSTATE_ENABLED and TBSTATE_PRESSED are in commctrl
```

We don't want this to be called each and every time – rather we want it to be called only when the pressed or unpressed state changes. Thus, we need a flag to keep track of when the status changes; the best place to put that flag is in the declaration of our `GWCommand`. Why? Because the `GWCommand` associated with our button is both persistent and easily accessible from our `Validate` method. Finally, let's add a property that holds the window handle of the button bar – why have to search for it each time? The same is true for a property holding the message's `MessageID`. What we are trying to do here is make `GWCommand` do the simple task of storing persistent data tied to a particular message view, rather than having to rediscover the information on each `Validate`; so we will add to our declaration, shown in the example below.

Delphi Example

```
Command = class(TGWCommand)
private
    { Private declarations }
    LongPrmt : string;
    ToolTp : string;
    BaseCommand : Variant;
    function GetLongPrompt : string;
    function GetToolTip : string;
        ButtonID: Integer; //remember that we added this to hold the
            button's toolbarid
    dirty:boolean; //this is our added flag to keep track of the depression state
    done:boolean; //this is a flag that will keep track of whether Validate has
        been run before
    window:thandle; //this will hold the handle to the button bar
    messageid:string // this will hold the MessageID of the message associated
        with the particular command
public
    m_nCmd : longint;
    Constructor Create(nCmd: longint);
automated
    property LongPrompt: string read GetLongPrompt;
    property ToolTip: string read GetToolTip;
    procedure Execute;
    function Validate: longint;
end;
```

We must be sure to set our flags off when we create the Command object:

Delphi Example

```
Constructor Command.Create(nCmd: longint);
begin
    inherited Create;
    m_nCmd := nCmd;
    done:=false;
    dirty:=false;
    window=0;
    messageid='';
end;
```

Our Execute method will set the dirty flag as necessary. This is a signal to the Validate that the button state has changed.

Delphi Example

```
procedure Command.Execute;
begin
    case m_nCmd of
        MYCOMMANDID:
            begin
                dirty:=not dirty;
                buttonon:=not button on; //this is is a global variable that
                    will hold the button state
            end;
    end;
end;
```

Note that we need buttonon, a global variable, to keep track of whether the button is pressed. We could also save the state in GWCommand, but that would make it more difficult to access from other procedures that must know the button's state. Another alternative would be to add a custom field to the message that keeps track of the button state.

Now, we can get to our Validate method where the heart of the work is done.

Delphi Example

```
function Command.Validate: longint;
var win,handle:tHandle;
begin
    if m_nCmd = MYCOMMANDID then begin
        if not done then //Only get the MessageID if we have not done so already
            for this Command. Note, we could also do:if messageid = ''
                to avoid the done flag. Or, we could have set the
                messageid during the Create method.
        begin
            Commander.Execute('itemmessageidfromview()',messageid);
            done:=true;
        end;
    end;
end;
```

```

end;
if messageid<>'X00' then //if not new message then disable item
    result:=eGW_CMDVAL_DISABLED
else //otherwise, let's press the button if necessary

if (window=0) then begin//get the handle to the toolbar if we don't have
    it
    win:=getforegroundwindow;
    enumchildwindows(win,@enumrichedit,0);
    enumchildwindows(win,@enumproc,longint(@handle));
    window:=handle;
end;
end;
if ButtonOn then //check if the button is on.
    this is unrelated to actually pressing the button
    if the button is on, we have to 'check' it -
    unfortunately we can't save cycles here

    result:=eGW_CMDVAL_CHECKED;
if dirty then //only press buttons if the user changed the state
begin
    dirty:=false; //reset so we don't check until user changes again
    if not ButtonOn then
        sendmessage(window,TB_setstate,toolbarid,longint(TBSTATE_ENABLED))
    else
        sendmessage(window,TB_setstate,toolbarid,longint
            (TBSTATE_PRESSED));
end;
end;
end;

```

Let's take a closer look at what we have done here. First, we get the MessageID if we haven't already. As discussed in the comments above, there are many ways that we can do this. Second, we disable the buttons if this is not a new message. Next, we get the buttonbar window handle if we don't have it. Finally, we press the button according to state, but only if the user recently changed. If there is no change by the user, no change by the `validate` code is necessary.

One final note about the button bar. If you want to remove, rather than just disable buttons on non-new messages, you will need to do that in the `CustomizeToolBar` method. The complexity is that tokens will not work, as the message view is not opened yet. So, we will use the `ClientState`, yet another tool:

Delphi Example

```

function CommandFactory.CustomizeToolBar( Context: string;
                                           GWToolBar: variant): wordbool;
var clistate,testmessage:variant;
begin
if (CompareText('GW.MESSAGE',Copy(context,1,10)) = 0) then begin
try
    clistate:=c3po.g_C3POManager.ClientState;
    testmessage := clistate.CommandMessage; //get the current message
except //exception gets thrown by clistate.commandmessage if it is a new message
    This is because new messages are not GW objects yet

```

```

Cmd := Command.Create(MYCOMMANDID);
Toolbar := ToolbarItems.Add('My Button', Cmd.OleObject);
Cmd.ButtonID:=Toolbar.toolbarid;
  PGPCmd.toolbar:=true;
Toolbar.SetBitmap(extractfilepath(paramstr(0))+ 'myicons.dll',1);
Cmd.ToolTp := 'Long prompt for my button';
Cmd.Release;
end;
end;

```

Note how we use the exception to determine the type of message we have. We aren't really deleting the button if it is a new message, we are simply not adding it. We could, of course, add other buttons in the try block. Those messages would be added to non-new messages.

The Message Field

We may want to disable menus based on what field the user is in. For example, we may want to process message text, but not the subject. We would need to know the field, however, if we were going to select text. Here is a simple algorithm in the `Validate` method to test whether you are in the message field:

Delphi Example

```

if messageid <> 'X00' then result:=eGW_CMDVAL_DISABLED else begin
  commander.execute('EnvTextCurrentLineIndex()',temp);
  if pos('Token failed',temp)>0 then
    result:=eGW_CMDVAL_DISABLED;
end;

```

Like the `ClientState.CommandMessage` above, here we are taking advantage of a token that will fail unless it is in the message view. This example and those above should hopefully give you an idea of the power of using all of the API interfaces available to you to creatively achieve your needs.

The Report Generator

The Report Generator is a C3PO that adds a menu to the “Tools” menu. The generator will search your calendar (or a proxied central calendar) for a code in the subject, and generate an ordered list of all calendar items with that code in the subject. A use for this is in a law office, where the code is a client code, and all appointments have a client code.

Use The Wizard

Our first task is to decide how we want to invoke the report generator. A simple menu item on the toolbar will do. We will be able to use the C3PO Wizard to create code that will make the menu item. The following is the output of the C3PO Wizard:

```
#C3PO WIZARD
#Define Menu=GW.CLIENT.WINDOW.BROWSER
#Begin
#MenuItem=MenuItem
#Menu=Tools
#Text=Generate a Docket Report
#LongPrompt=Generate a report of all appointments and tasks with a specific
string in the subject..
#Constant=GENREP
#End
#Define Menu=GW.MESSAGE
#Begin
#MenuItem=MenuItem
#Menu=Tools
#Text=Generate Docket Report
#LongPrompt=Generate a report of all appointments and tasks with a given string
in the subject.
#Constant=GENREP
#End
#Done
```

Let's take a closer look at what we have done:

```
#Define Menu=GW.CLIENT.WINDOW.BROWSER
#Define Menu=GW.MESSAGE
```

These lines tell the wizard code generator to add menus to the browser and to all message views.

```
#Constant=GENREP
```

This is the constant name that will be included in our new `GWCommand` object and tested in our `GWCommand.Execute` method. Note that the code generator will automatically assign an integer value to this constant name. Note also that I used the same constant name for both the browser and the message instantiation. This means that the same command will be called, regardless of where the menu item sits.

```
#MenuItem=MenuItem
#Menu=Tools
```

These lines tell the user what type of menu object to make and where to put it.

```
#Text=Generate Docket Report
#LongPrompt=Generate a report of all appointments and tasks with a given string
in the subject.
```

These lines define the user interface portions of the menus. Let's assume that the C3PO wizard has now generated code. We will look at that code later.

The Form

Unrelated to GroupWise but important nonetheless is the form we will use to get user information. Here, we have a form with a couple of hidden fields to do data processing.

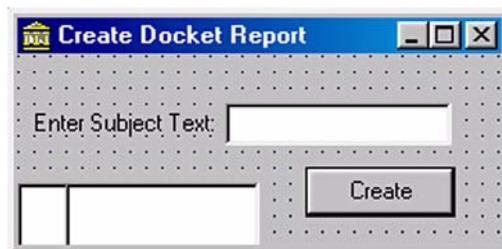


Figure 1: *Create Docket Report.*

The form actually does most of the work. All our C3PO needs to do is open the form. However, you need to call

```
Application.ShowMainForm:=false;
```

before

```
Application.CreateForm(TForm1, Form1);
```

so that the form will stay hidden until your C3PO opens it.

Delphi Example

Here is the actual code for the form:

```
object Form1: TForm1
  Left = 351
  Top = 257
  Width = 250
  Height = 122
  Caption = 'Create Docket Report'
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET
```

```

Font.Color = clWindowText
Font.Height = -11
Font.Name = 'MS Sans Serif'
Font.Style = []
OldCreateOrder = False
PixelsPerInch = 96
TextHeight = 13
object Label1: TLabel
    Left = 8
    Top = 27
    Width = 91
    Height = 13
    Caption = 'Enter Subject Text:'
end
object SearchString: TEdit
    Left = 104
    Top = 24
    Width = 113
    Height = 21
    TabOrder = 0
end
object Create: TButton
    Left = 144
    Top = 56
    Width = 75
    Height = 25
    Caption = 'Create'
    Default = True
    TabOrder = 1
    OnClick = CreateClick
end
object TheDates: TListBox
    Left = 0
    Top = 64
    Width = 81
    Height = 81
    ItemHeight = 13
    TabOrder = 2
    Visible = False
end
object TheDisplay: TListBox
    Left = 24
    Top = 64
    Width = 97
    Height = 41
    ItemHeight = 13
    TabOrder = 3
    Visible = False
end
end
end

```

The C3PO Server

The C3PO server in this project needs no real modification. The C3PO Wizard will generate this code, but let's take a look at some of the key sections.

CanShutdown

You want to make sure that you tell GroupWise that it can shut down when it wants to. Thus, your code should include this.

Delphi Example

```
function C3POServer.CanShutdown: ToleBool;
begin
    Result := TRUE;
end;
```

RegisterServer

You will need to get your server registered. This is done using the /regsrvr startup parameter. Also, if you create an executable, you can run the executable once, and it will automatically register the information. The wizard kindly puts this in the code for you:

Delphi Example

```
function RegisterServer : HRESULT;
var
    Reg : TRegistry;
    sRegKeyName : string [120];
    sAppName : string [120];

begin
    sAppName := 'repgen';
    Reg := TRegistry.Create;
    Reg.RootKey := HKEY_LOCAL_MACHINE;
    sRegKeyName :=
        '\SOFTWARE\Novell\GroupWise\5.0\C3PO\DataTypes\GW.CLIENT.WINDOW.BROWSER
        \repgen';
    Reg.OpenKey (sRegKeyName, TRUE);
    Reg.OpenKey ('Objects', TRUE);
    Reg.WriteString ('CommandFactory', '');
    Reg.OpenKey (sRegKeyName, TRUE);
    Reg.OpenKey ('Events', TRUE);

    sRegKeyName :=
        '\SOFTWARE\Novell\GroupWise\5.0\C3PO\DataTypes\GW.MESSAGE\repgen';
    Reg.OpenKey (sRegKeyName, TRUE);
    Reg.OpenKey ('Objects', TRUE);
    Reg.WriteString ('CommandFactory', '');
    Reg.OpenKey (sRegKeyName, TRUE);
    Reg.OpenKey ('Events', TRUE);

    Result := S_OK;
end;
```

Further, we need to get the OLE information registered as well. The C3PO code generator also generates this code:

Delphi Example

```
procedure RegisterC3POServer;
const
  AutoClassInfo: TAutoClassInfo = (
    AutoClass: C3POServer;
    ProgID: 'repgen';
    ClassID: '{B638DF40-8410-11D3-82BD-0090274773AE}';
    Description: 'C3po Automation';
    Instancing: acMultiInstance);
begin
  Automation.RegisterClass(AutoClassInfo);
  RegisterServer;
end;
```

Initializing

In the initialization sequence, we will set up the Token Commander and the ObjectAPI.

Delphi Example

```
procedure C3POServer.Init(Manager: variant);
begin
  g_C3POManager := Manager;
  AccountLogon;
  commander:=CreateOleObject('GroupWiseCommander');
  OleInitialize(nil);
end;
```

Our AccountLogon procedure looks like this:

Delphi Example

```
procedure AccountLogon;
  var Groupwise,Temp:variant;
begin
try
  GroupWise:=CreateOleObject('NovellGroupWareSession');
  temp:=GroupWise.Login;
  try
  GWAccount:=GroupWise.Proxy('Library');
  except GWAccount:=temp; end;
  except end;
end;
```

Because the Token Commander and Object API must be accessible from our main C3PO code, we have declared `GWAccount` and `commander` as global variables in the main form's unit, and you will add that unit to the main form's uses clause.

```
uses
. . .
repgenC3PO,
repgenUnit,
. . .;
```

De-Initializing

It is important to de-initialize your C3PO. Note that we have added to the default code to release the form and the application.

Delphi Example

```
procedure C3POServer.DeInit;
begin
  g_C3POManager := Unassigned;
  OleUninitialize();
  Form1.Release;
  Application.Terminate;
end;
```

The C3PO Object

Let's take a look at the actual C3PO, namely the implementation of the menus.

The CommandID Constant

First, note the following line:

```
Const

  GENREP = 0;
```

This is where `GENREP`, from the wizard above, is now defined as an actual constant. We will use this later.

Initializing the C3PO

This is where you tell GroupWise what your C3PO plans to do with the GroupWise client. Here, all we want to do is modify the menus.

Delphi Example

```
function CommandFactory.Init(lcid : longint): longint;
begin
  result := eGW_CMDINIT_MENU;
end;
```

Adding the Menu

Now we will add the menu. GroupWise will call our `CustomizeMenu` method whenever it needs to create a new menu. We have left the comments by the C3PO Wizard in, as they explain exactly how the process proceeds.

Delphi Example

```
function CommandFactory.CustomizeMenu( Context: string;
                                       GWMenu: variant): ToleBool;

var
  vMenuItems : variant;
  vMenu : variant;
  vSeparator: variant;
  Cmd : Command;
begin

  If CompareStr(Context, 'GW.CLIENT.WINDOW.BROWSER') = 0 then           // Check
for correct context
  begin
    vMenu := GWMenu;           // get menu form GWMenu
    vMenu := vMenu.MenuItems.Item('Tools');           // get menu Tools
    GwCmdGENREP := Command.Create(GENREP);           // create command for
      Custom menu
    vMenuItems := vMenu.MenuItems.Add('Generate a Docket Report',
      GwCmdGENREP.OleObject);           // add menu item to the
      end of the menu
    GwCmdGENREP.LongPrmt := 'Generate a report of all appointments and tasks
      with a specific string in the subject..';           // set
      long prompt for menu item
    GwCmdGENREP.Release;

  end;
  If Pos('GW.MESSAGE', Context) <> 0 then           // see if the context is
      GW.MESSAGE or any sub class
  begin
    vMenu := GWMenu;           // get menu form GWMenu
    vMenu := vMenu.MenuItems.Item('Tools');           // get menu Tools
    GwCmdGENREP := Command.Create(GENREP);           // create command for
      Custom menu
    vMenuItems := vMenu.MenuItems.Add('Generate Docket Report',
      GwCmdGENREP.OleObject);           // add menu item to the end of the
      menu
    GwCmdGENREP.LongPrmt := 'Generate a report of all appointments and tasks
      with a given string in the subject.';           // set long prompt for
      menu item
    GwCmdGENREP.Release;

  end;

  result := FALSE; // this is a non-volatile menu

end;
```

Essentially, this method is testing for which context (e.g. window) is getting the menu, and then it adds a new menu item to the “Tools” menu. Note how we release the `GWCommand` we created after we have passed it to `GroupWise`. The command does not disappear, however, because `GroupWise` still maintains a reference to it. You could maintain your own reference, as well, if you wanted.

The Execute Method

Now we need to define our `Execute` method, so that our program actually does something when the user selects our new menu item.

Delphi Example

```
procedure Command.Execute;
begin

    case m_nCmd of // this is where we test for our CommandID

        GENREP:
            begin
                Form1.Show;
                SetForegroundWindow(Form1.Handle);
                Form1.SearchString.SetFocus;
            end;

        -1: //Not used
        else
            MessageBox(0, 'No Command ID found?', 'Execute Command',
                MB_SETFOREGROUND); // do a message box
    end;
end;
```

Note how simple this `Execute` method is; all we do is open our form, move it to the front, and focus on the input window. If you have multiple commands, you will have more `CommandID` constants to check.

The Validate Method

Because our menu will always be activated, our `Validate` method will be simple.

Delphi Example

```
function Command.Validate: longint;
begin
    result := eGW_CMDVAL_ALWAYS;
end;
```

However, we could do checking to enable or disable the menu.

The Form Functions

This seems like a lot of wind-up to get to the actual meat of the program. You will find that the C3PO Wizard automates so much that you will actually be able to get right to the key parts of the program.

The Create Button

Nothing happens in our form until the `Create` button is pressed, so let's start with this button.

Delphi Example

```
procedure TForm1.CreateClick(Sender: TObject);
begin
    CreateReport;
    Form1.Hide;
end;
```

Simple enough. So here is `CreateReport`:

Delphi Example

```
procedure CreateReport;
var FoundMessages:variant;
    CurrentMessage:variant;
    result:string;
    count,max:integer;
    commandstring:string;
    TheDate:string;
begin
    FoundMessages:=GWAccount.AllMessages.Find('(BOX_TYPE = INCOMING) AND (SUBJECT
CONTAINS ''+Form1.SearchString.Text+') AND (TASK OR APPOINTMENT) AND
(START_DATE >= TODAY)');{ }
    SortItems(FoundMessages);
    commander.execute('NewMail()',result);

    for count := 0 to Form1.TheDisplay.Items.Count-1 do
    begin
        commander.execute(Form1.TheDisplay.Items[Count],result);
    end;
    Form1.SearchString.Text:='';
    Form1.Hide;
end;
```

Executing the Query

Here we get our first taste of the ObjectAPI. Because we logged in during the initialization of the C3PO server, we do not need to log in here – `GWAccount` already points to a valid `Account` object. In this procedure we search for all incoming appointments and tasks, that contain our search string in the subject field. We also want to find only messages after today.

Sorting the Items

Delphi Example

Then we call `SortItems`, a procedure that will put the messages in order. We pass in `FoundMessages` which is our `MessageList` object that contains our found messages.

```
procedure SortItems(Messages:Variant);
var CurrentMessage:variant;TheDate:string;Count,ListCount:Integer;
Date:boolean;
begin
Form1.TheDisplay.Clear;
Form1.TheDates.Clear;
If Messages.Count > 0 then
  begin
    CurrentMessage:=Messages.Item(1);
    try
    If CurrentMessage.ClassName = 'GW.MESSAGE.APPOINTMENT' then
      TheDate:=CurrentMessage.StartDate
    else
      TheDate:=CurrentMessage.DueDate;
    except TheDate:='Message';end;
    Form1.TheDisplay.Items.Add(format('ItemSetText("X00";Message!;"%s %s
%s";Yes!)',[TheDate,CurrentMessage.Subject,chr(10)]));
    Form1.TheDates.Items.Add(TheDate);
    For Count := 2 to Messages.Count do
    begin
      CurrentMessage:=Messages.Item(Count);
      try
      If CurrentMessage.ClassName = 'GW.MESSAGE.APPOINTMENT' then
        TheDate:=CurrentMessage.StartDate
      else
        TheDate:=CurrentMessage.DueDate;
      except TheDate:='Message';end;

      ListCount:=0;
      While (ListCount < (Form1.TheDates.Items.Count)) do
        begin
          if
            StrToDateTime(TheDate) >
            StrToDateTime(Form1.TheDates.Items[ListCount])
          then
            ListCount:=ListCount+1
          else
            break;
        end;
      Form1.TheDates.Items.Insert(ListCount,TheDate);
      Form1.TheDisplay.Items.Insert(ListCount,
        format('ItemSetText("X00";Message!;"%s %s
%s";Yes!)',[TheDate,CurrentMessage.Subject,chr(10)]));
    end;
  end;
end;
```

This procedure is doing the bulk of the work. We use the two hidden fields on the form to hold data; we could have used a linked list, or a `tstringlist` as well. The first thing we do is count how many messages were found – we don't want to act if there were no found messages.

Then we iterate through the messages one by one. We check to see if the message is an appointment or a task. We do this using the ObjectAPI. The type of message is important because the relevant date will either be the start date of the appointment or the due date of the task. We set a date variable equal to these dates. Then we go through the dates we have and do a compare to insert the date in the right place. Note that the date is supposed to be a `tdatetime`, but for some reason they did not compare here – they were returned as strings. Thus, we convert the date string to a `tdatetime` for comparison.

Finally, note how we add not only the date to the hidden listbox, but instead we add the entire `ItemSetText()` token to the box, so that at a later time we can send it directly to the token commander.

Creating the New Message

Then, we use the token commander to open a new mail.

Delphi Example

```
commander.execute('NewMail()',result);

for count := 0 to Form1.TheDisplay.Items.Count-1 do
begin
    commander.execute(Form1.TheDisplay.Items[Count],result);
end;
Form1.SearchString.Text:='';
Form1.Hide;
```

Here we call the `NewMail()` token, and then we scroll through our hidden listbox and send the token commands we created earlier to the token commander.

Summary

That is the whole project – it uses the Token Commander, ObjectAPI and C3PO's in a relatively simple project. It is a good example of how to put all of these interfaces together.

Copyright © 2002 by Novell, Inc. All rights reserved.
No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, for any purpose without the express written permission of Novell.

All product names mentioned are trademarks of their respective companies or distributors.

Novell, Inc.

464-000063-013

GroupWise® Developer's Guide

by Michael Risch, Sean Kirkby,
Bob Good, & Steve Hughes

www.novell.com/research

Novell®

Novell, Inc.
1800 Novell Place
Provo, Utah 84606
USA