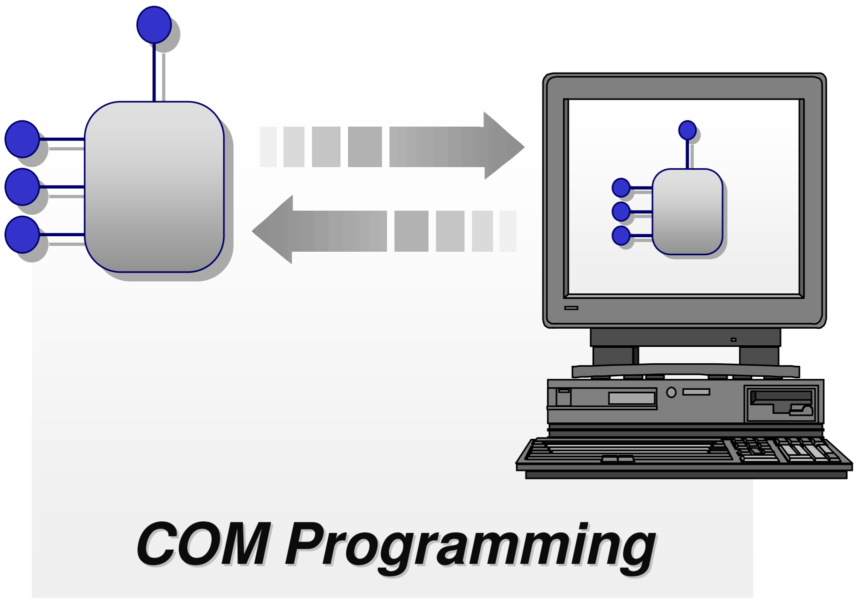
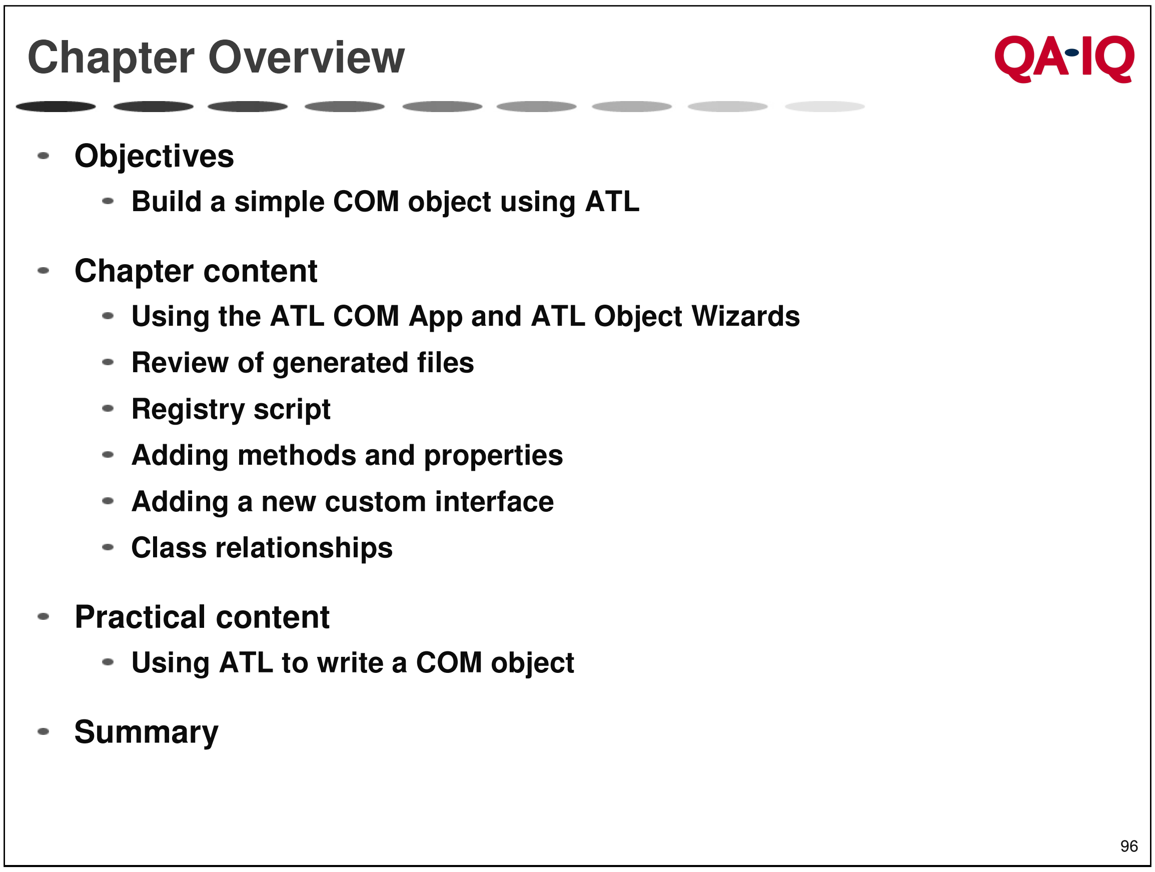
Building a COM Object using ATL

QA-IQ

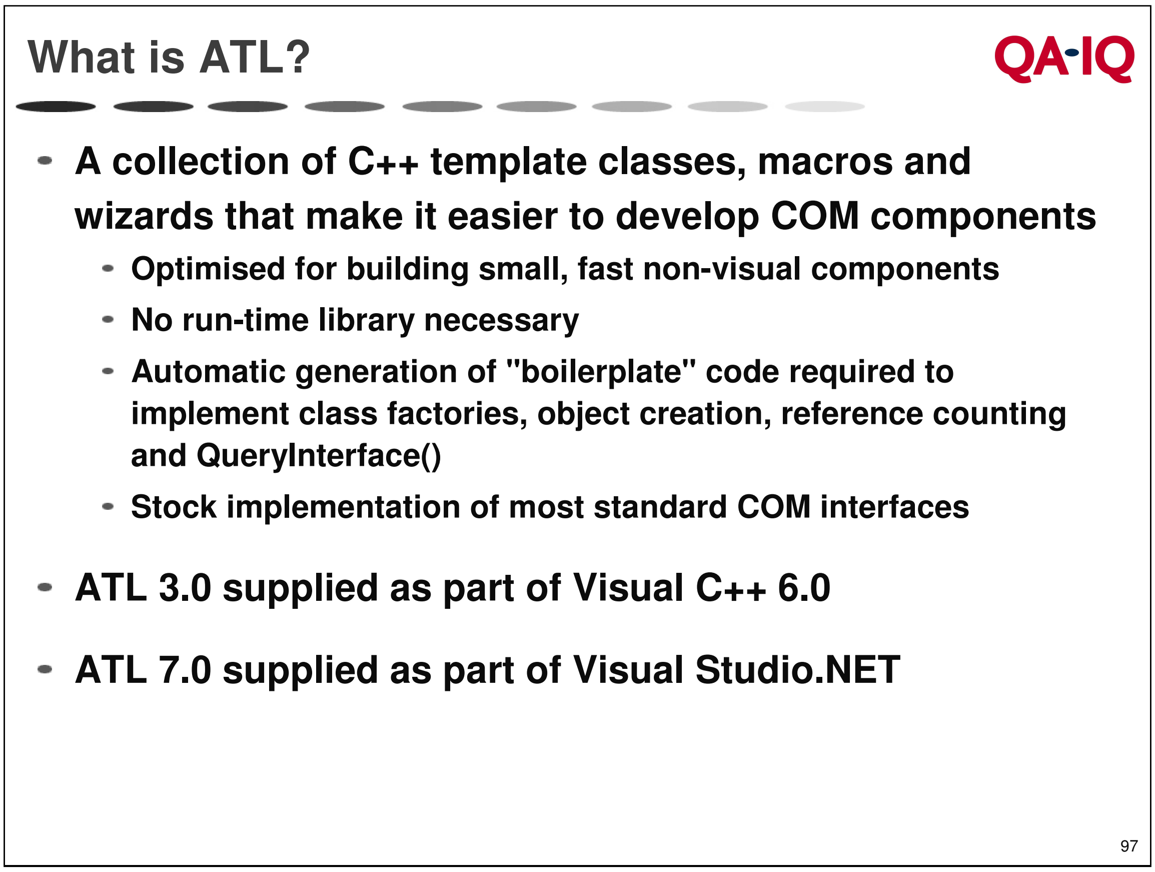


95



Chapter Overview

- **Objectives**
  - **Build a simple COM object using ATL**
- **Chapter content**
  - **Using the ATL COM App and ATL Object Wizards**
  - **Review of generated files**
  - **Registry script**
  - **Adding methods and properties**
  - **Adding a new custom interface**
  - **Class relationships**
- **Practical content**
  - **Using ATL to write a COM object**
- **Summary**



What is ATL?

- A collection of C++ template classes, macros and wizards that make it easier to develop COM components
  - Optimised for building small, fast non-visual components
  - No run-time library necessary
  - Automatic generation of "boilerplate" code required to implement class factories, object creation, reference counting and QueryInterface()
  - Stock implementation of most standard COM interfaces
- ATL 3.0 supplied as part of Visual C++ 6.0
- ATL 7.0 supplied as part of Visual Studio.NET

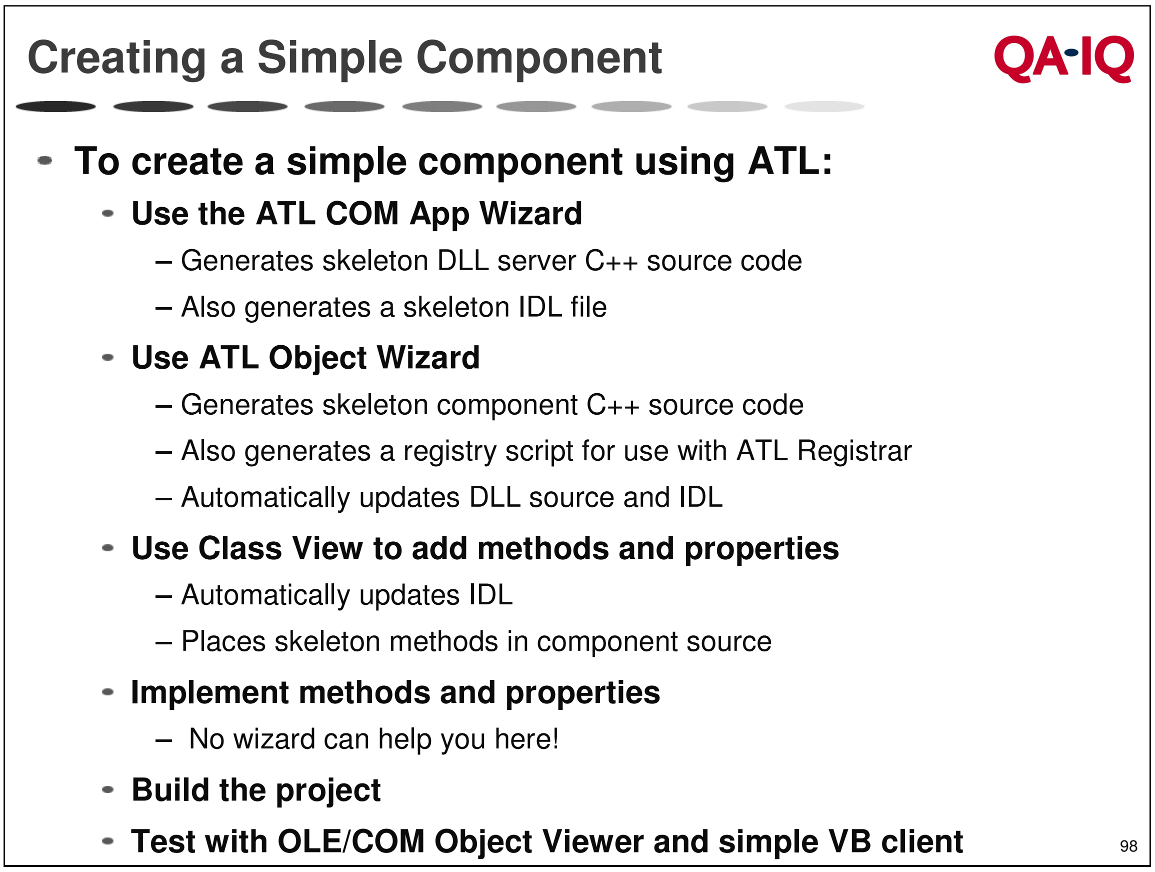
97

The Active Template Library (ATL) is now the preferred method for building COM objects. We will be using version 3.0, which is supplied with Visual C++ 6.0.

ATL version 7.0 is provided with Visual Studio.NET. This is broadly speaking compatible with ATL 3.0, but with notable changes around the CComModule class and with considerably improved support classes.

There is also a new set of native C++ classes designed to help you author web applications, collected under the name *ATL Server*.

One other significant change with ATL 7.0 and Visual Studio.NET is that the C++ compiler supports attributes directly. In combination with ATL, this obviates the need to use IDL for much of your COM programming.



Creating a Simple Component

- To create a simple component using ATL:
  - Use the ATL COM App Wizard
    - Generates skeleton DLL server C++ source code
    - Also generates a skeleton IDL file
  - Use ATL Object Wizard
    - Generates skeleton component C++ source code
    - Also generates a registry script for use with ATL Registrar
    - Automatically updates DLL source and IDL
  - Use Class View to add methods and properties
    - Automatically updates IDL
    - Places skeleton methods in component source
  - Implement methods and properties
    - No wizard can help you here!
  - Build the project
  - Test with OLE/COM Object Viewer and simple VB client

98

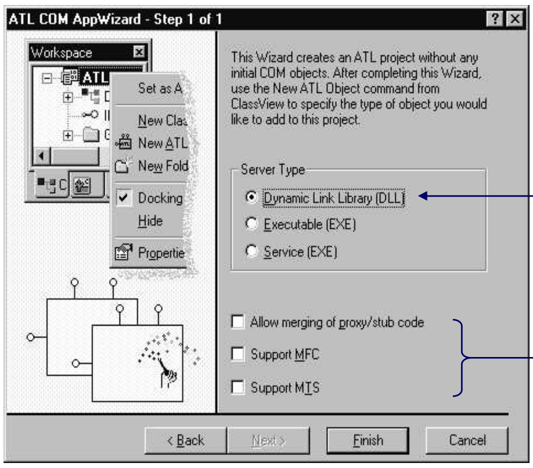
In the following slides, we will create a simple Account component using the ATL wizards provided with Visual C++ 6.0.

Note that when creating a new project for our component, the name specified will become the name of the server DLL (module), not the name of a component. Because these names must be different, we've called our project *SimpleAccount*, but we could have called it *AccountServer* or *AccountModule*.

Using the ATL COM AppWizard

QA-IQ

- In Visual Studio , create new ATL COM APP Wizard project
  - Accept defaults (DLL server type, etc.), then click Finish



ATL COM AppWizard - Step 1 of 1

This Wizard creates an ATL project without any initial COM objects. After completing this Wizard, use the New ATL Object command from ClassView to specify the type of object you would like to add to this project.

Server Type
- (●) Dynamic Link Library (DLL)
- ( ) Executable (EXE)
- ( ) Service (EXE)

- [ ] Allow merging of proxy/stub code
- [ ] Support MFC
- [ ] Support MTS

For the purposes of this exercise, choose DLL server type

For the purposes of this exercise, leave all unchecked

99

In the *ATL COM AppWizard - Step 1 of 1* dialog, we'll simply accept the defaults to create a DLL server. We'll look at building an EXE server in a later chapter.

After clicking *Finish*, the wizard displays a *New Project Information* dialog that lists some of the files that will generated when we click *OK*. We can see a complete list by looking at the *FileView* of the new project.

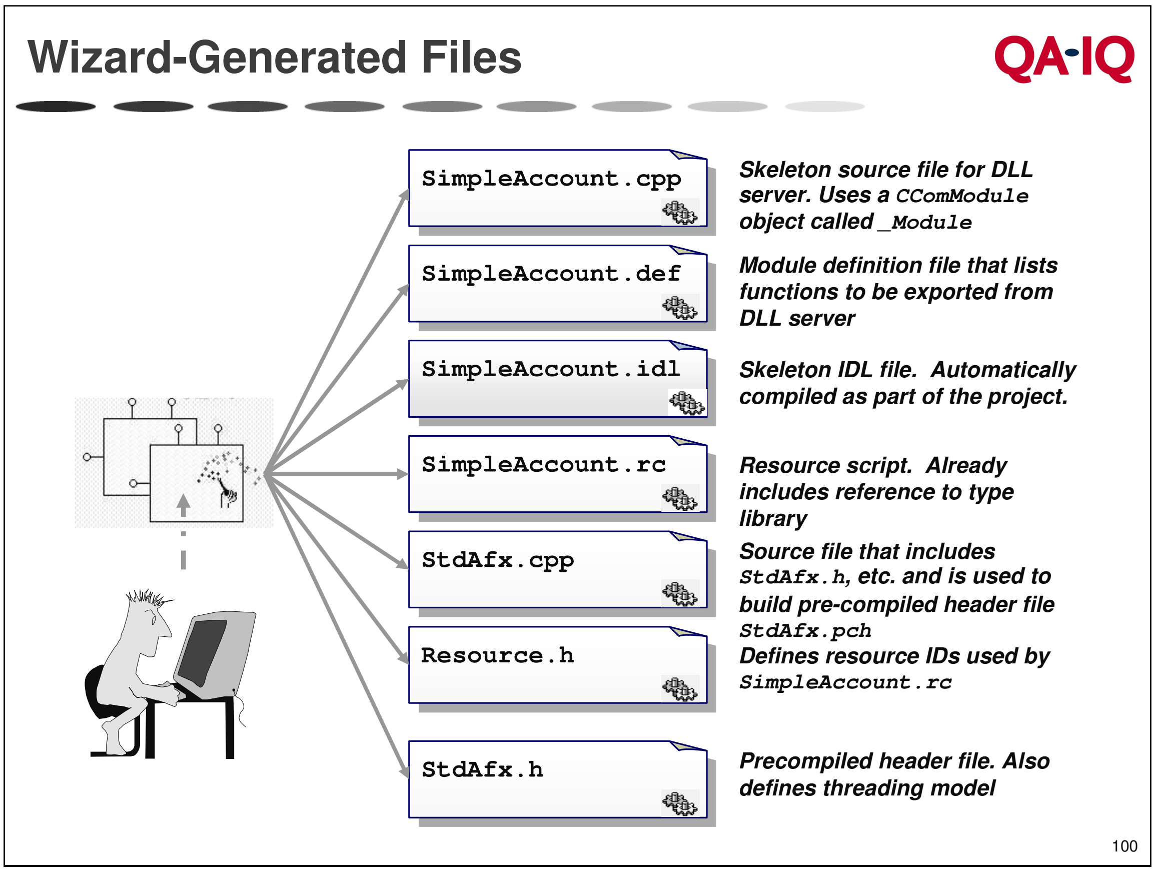
The bottom three options allow you to

(a) add your proxy/stub code to your DLL (more on proxies and stubs later);

(b) support MFC (don't do it!)*, or

(c) add support for registering your components with Microsoft Transaction Services (MTS).

* Why the "don't do it!" comment? ATL was originally designed for writing lean, small footprint ActiveX controls. Checking the *Support MFC* option, whilst enabling use of constructs like `CString`, adds a dependency to the MFC runtime DLL. ATL now provides capabilities for string manipulation, windowing and so on. Therefore, do you still *really* need MFC?



Wizard-Generated Files

QA-IQ

`SimpleAccount.cpp`	*Skeleton source file for DLL server. Uses a `CComModule` object called `_Module`*
`SimpleAccount.def`	*Module definition file that lists functions to be exported from DLL server*
`SimpleAccount.idl`	*Skeleton IDL file. Automatically compiled as part of the project.*
`SimpleAccount.rc`	*Resource script. Already includes reference to type library*
`StdAfx.cpp`	*Source file that includes `StdAfx.h`, etc. and is used to build pre-compiled header file `StdAfx.pch`*
`Resource.h`	*Defines resource IDs used by `SimpleAccount.rc`*
`StdAfx.h`	*Precompiled header file. Also defines threading model*

100

Here is a complete list of the files that are generated by the *ATL COM AppWizard*. We will look at the primary C++ source file, `SimpleAccount.cpp`, over the next three slides.

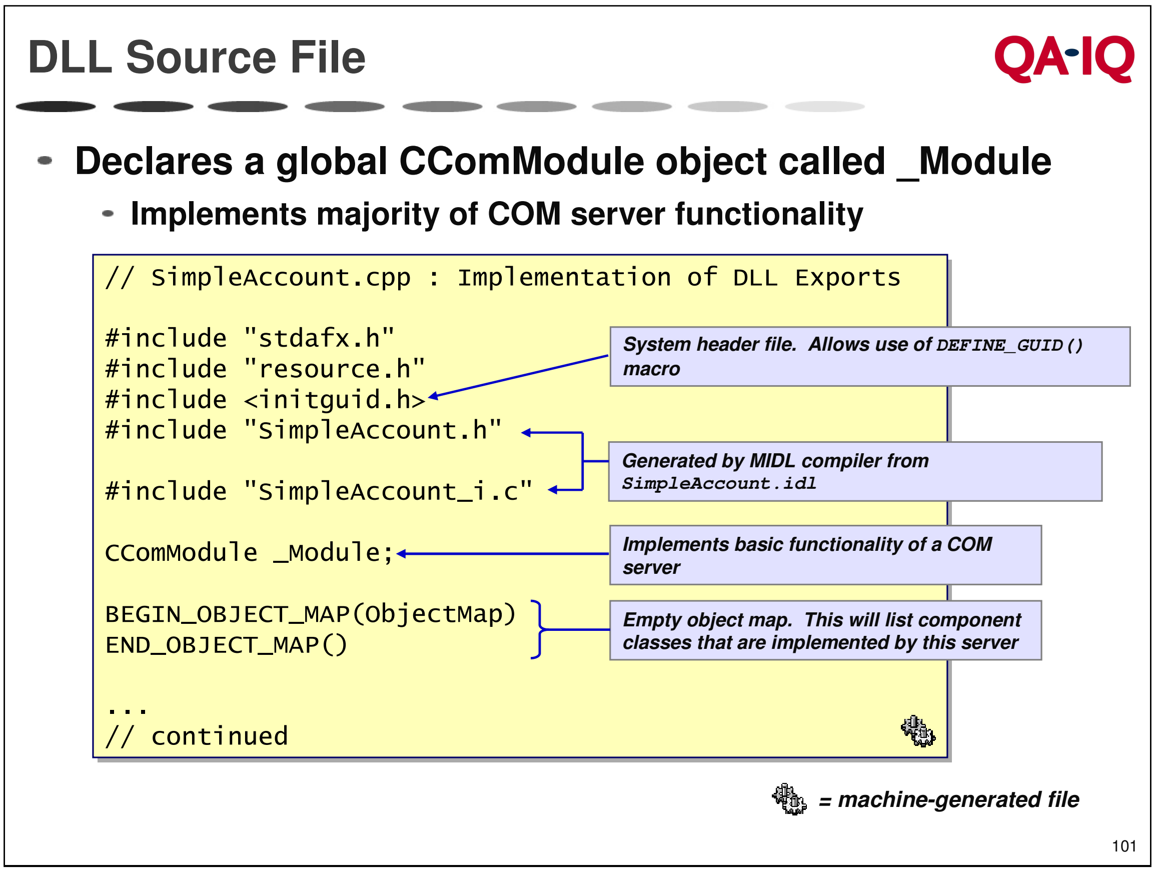
The module definition file, `SimpleAccount.def`, contains the by-now familiar exports list for a DLL server:

```
LIBRARY      "SimpleAccount.DLL"

EXPORTS
    DllCanUnloadNow       @1 PRIVATE
    DllGetClassObject     @2 PRIVATE
    DllRegisterServer     @3 PRIVATE
    DllUnregisterServer   @4 PRIVATE
```

The IDL file, `SimpleAccount.idl`, contains the skeleton of a `library` statement for creating a type library:

```
import "oaidl.idl";
import "ocidl.idl";
[
  uuid(B7713F00-F3DA-11D2-9C70-000000000000),
  version(1.0),
  helpstring("SimpleAccount 1.0 Type Library")
]
library SIMPLEACCOUNTLib
{
  importlib("stdole32.tlb");
  importlib("stdole2.tlb");
};
```

Note that the imported files, `oaidl.idl` and `ocidl.idl`, define automation and standard COM interfaces, respectively. Similarly, the imported type libraries, `stdole32.tlb` and `stdole2.tlb`, contain information on automation types.



DLL Source File

QA-IQ

- **Declares a global CComModule object called _Module**
  - **Implements majority of COM server functionality**

```
// SimpleAccount.cpp : Implementation of DLL Exports

#include "stdafx.h"
#include "resource.h"
#include <initguid.h>
#include "SimpleAccount.h"

#include "SimpleAccount_i.c"

CComModule _Module;

BEGIN_OBJECT_MAP(ObjectMap)
END_OBJECT_MAP()

...
// continued
```

System header file. Allows use of `DEFINE_GUID()` macro

Generated by MIDL compiler from `SimpleAccount.idl`

Implements basic functionality of a COM server

Empty object map. This will list component classes that are implemented by this server

⚙ *= machine-generated file*

101

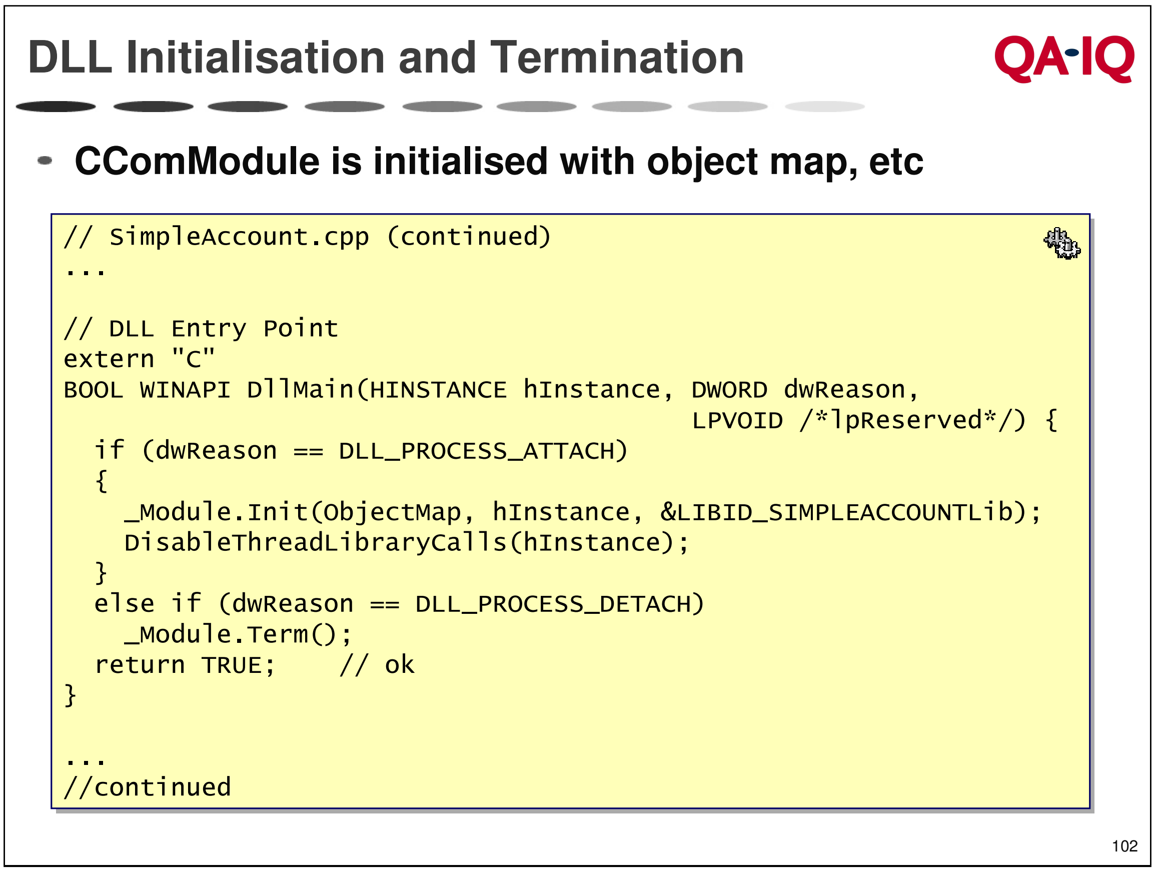
The primary C++ source file, `SimpleAccount.cpp`, includes a number of header files. `initguid.h` permits the use of the DEFINE_GUID() macro to initialise GUIDS, although it is not used in the generated code. Both `SimpleAccount.h` and `SimpleAccount_i.c` are generated by the MIDL compiler from `SimpleAccount.idl`. Therefore, these files won't actually exist until we build the project for the first time. (As you may recall from the IDL chapter, `SimpleAccount_i.c` will contain the GUID definitions.)

Next comes the declaration of a global instance of `CComModule` called `_Module`. `CComModule` implements the majority of the code required by either a DLL or EXE COM server.

This is followed by an empty *object map*, which will be filled in later by the *ATL Object Wizard*. Basically, this an array of `_ATL_OBJMAP_ENTRY` elements, where each element will contain the C++ class name of a component and its corresponding CLSID.

The global `CComModule` object uses this map to register objects in the Registry, to create objects (through an associated class factory), and to manage the lifetime of those objects.

The remainder of this source file contains the implementations of `DllMain()` and the exported DLL functions, as shown on the next two slides.



DLL Initialisation and Termination

QA-IQ

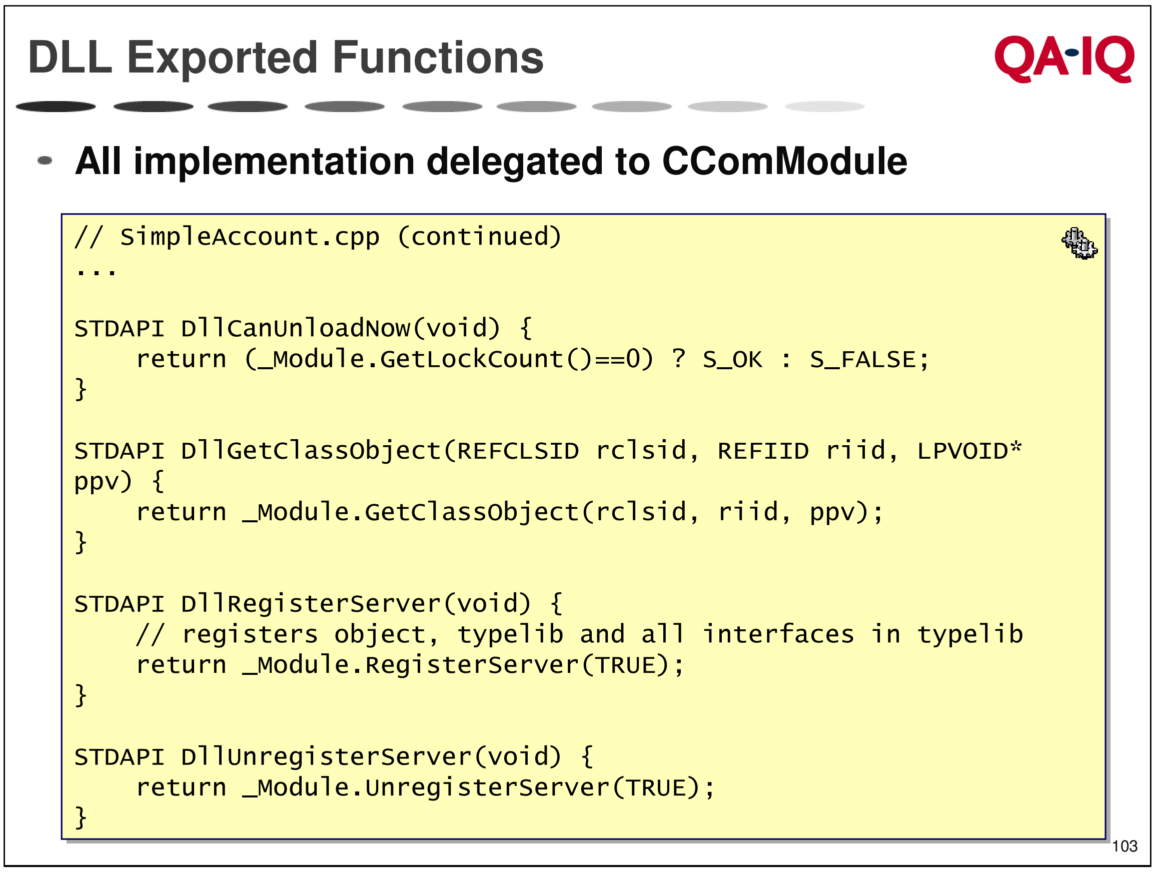
- **CComModule is initialised with object map, etc**

```
// SimpleAccount.cpp (continued)
...

// DLL Entry Point
extern "C"
BOOL WINAPI DllMain(HINSTANCE hInstance, DWORD dwReason,
                                         LPVOID /*lpReserved*/) {
  if (dwReason == DLL_PROCESS_ATTACH)
  {
    _Module.Init(ObjectMap, hInstance, &LIBID_SIMPLEACCOUNTLib);
    DisableThreadLibraryCalls(hInstance);
  }
  else if (dwReason == DLL_PROCESS_DETACH)
    _Module.Term();
  return TRUE;    // ok
}

...
//continued
```

102

As Win32 programmers, we know that `DllMain()` is the standard entry point of a Win32 DLL, and it can be called many times during the lifetime of that DLL. The second parameter of `DllMain()` indicates the reason for calling, which can be one of four values: `DLL_PROCESS_ATTACH`, `DLL_PROCESS_DETACH`, `DLL_THREAD_ATTACH` and `DLL_THREAD_DETACH`.

If `DllMain()` is called with `DLL_PROCESS_ATTACH`, it indicates that a new process has attached to the DLL, so any necessary per-process initialisation should be performed at this time. As shown on the slide, in `SimpleAccount.cpp`, the `Init()` function of the `CComModule` is called, passing the object map, the DLL module handle and the type library ID. Also, as with most DLLs, per-thread notification is disabled (for performance reasons) by calling `DisableThreadLibraryCalls()`.



DLL Exported Functions

QA-IQ

- **All implementation delegated to CComModule**

```
// SimpleAccount.cpp (continued)
...

STDAPI DllCanUnloadNow(void) {
    return (_Module.GetLockCount()==0) ? S_OK : S_FALSE;
}

STDAPI DllGetClassObject(REFCLSID rclsid, REFIID riid, LPVOID*
ppv) {
    return _Module.GetClassObject(rclsid, riid, ppv);
}

STDAPI DllRegisterServer(void) {
    // registers object, typelib and all interfaces in typelib
    return _Module.RegisterServer(TRUE);
}

STDAPI DllUnregisterServer(void) {
    return _Module.UnregisterServer(TRUE);
}
```

103

We should immediately recognise the last four functions in `SimpleAccount.cpp` as the functions that any DLL COM server should export: `DllGetClassObject()`, `DLLCanUnloadNow()`, `DllRegisterServer()` and `DllUnregisterServer()`. We discussed the implementation of these functions in depth in the *COM Fundamentals (Server)* chapter.

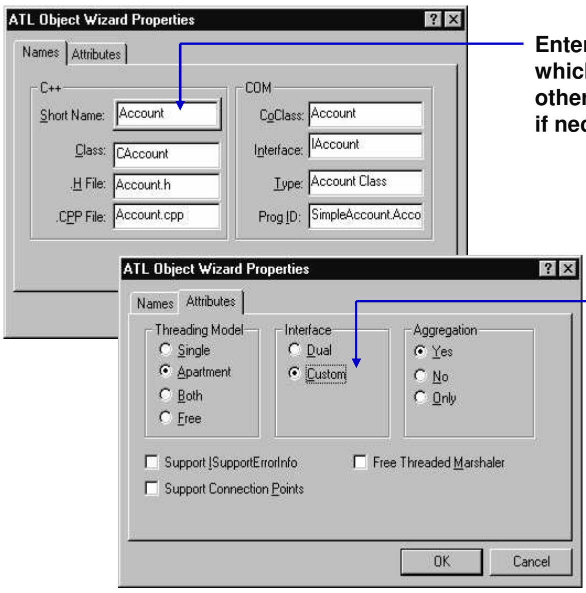
It's also no surprise to see that the actual implementation of these functions is delegated to the global `CComModule` object.

Although this server doesn't do anything useful yet, we can still build the project to see the new files generated by the MIDL compiler: `SimpleAccount.h`, `SimpleAccount_i.c` and `SimpleAccount.tlb`. These will then be listed under the *External Dependencies* folder in *FileView*.

Using the ATL Object Wizard

QA-IQ

- **After selecting Simple Object and clicking Next ...**



C++
Short Name: Account
Class: CAccount
.H File: Account.h
.CPP File: Account.cpp

COM
CoClass: Account
Interface: IAccount
Type: Account Class
Prog ID: SimpleAccount.Acco

Threading Model: Single [✓ Apartment] Both Free

Interface: Dual [✓ Custom]

Aggregation: [✓ Yes] No Only

☐ Support ISupportErrorInfo
☐ Support Connection Points
☐ Free Threaded Marshaler

Enter *Short Name* of Object, which serves as root for all other names. Can change these if necessary

For the purposes of this exercise, choose *Custom* interface. Otherwise, accept defaults

104

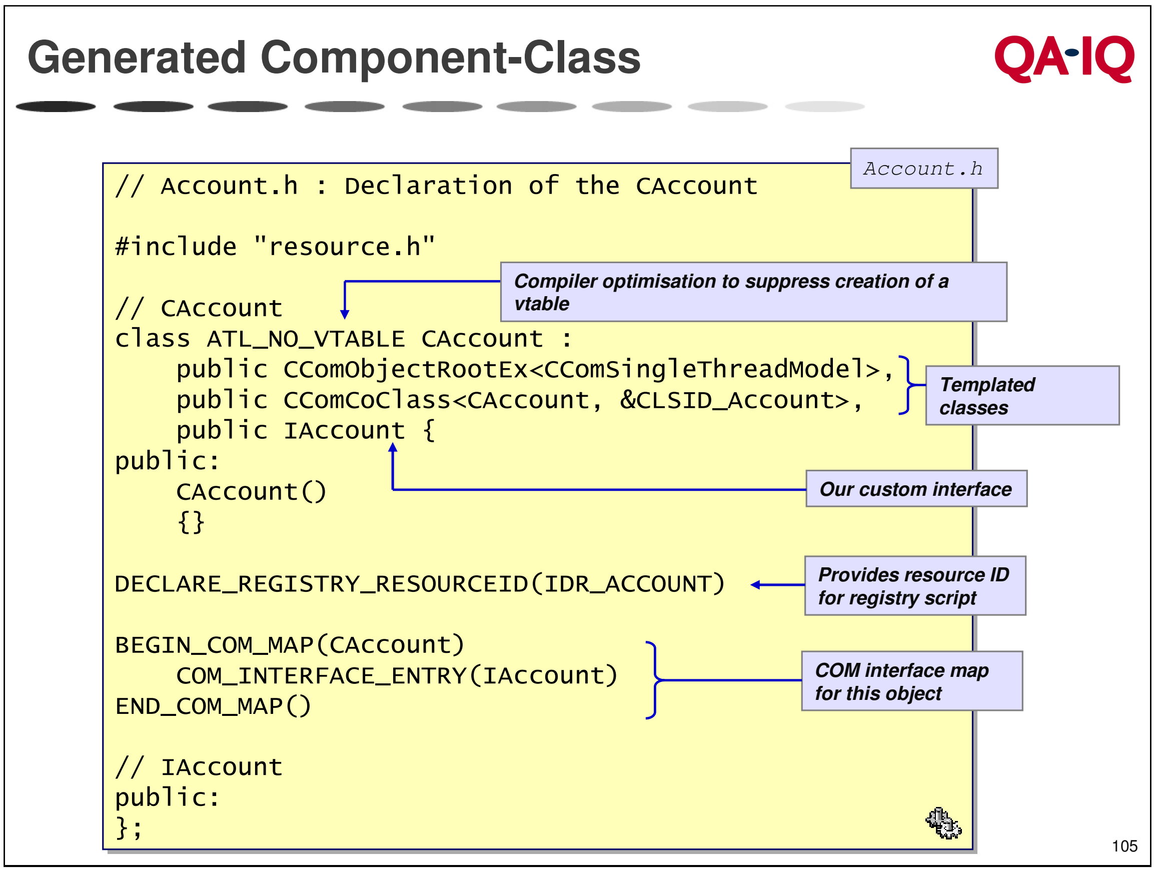
The next step is to add a simple COM object to our DLL server. Using the *ATL Object Wizard*, this is very easy. Selecting *New ATL Object…* from the *Insert* menu pops up a dialog that allows us to specify what type of object to create; all we need is a *Simple Object*. Clicking the *Next* button causes the *ATL Object Wizard Properties* sheet to be displayed. This has two tabs, *Names* and *Attributes*, as shown on the slide.

On the *Names* page, we need to type in a *Short Name* for the object; we've chosen *Account*. This name serves as a root for all the other names, but we can change these if we wish. For example, we might want to change the name of our custom interface. Note that, by default, the Prog ID of our COM object will simply be *SimpleAccount.Account*.

On the *Attributes* page, we would normally accept the defaults. However, as we've yet to learn about Automation and dual interfaces, we've chosen a *Custom* interface.

As to the meaning of the other options, *Apartment* threading model means that the COM object can be used by multiple threads, but the COM runtime will ensure that only the thread that created the object will ever call methods of the object. In other words, the COM object itself does not need to be thread safe. *Aggregation* is a method of reusing COM objects by containment (i.e. nesting one COM object inside another). The default *Aggregation* option of *Yes* means that this object can be aggregated and that ATL will implement `IUnknown` accordingly. `ISupportErrorInfo` is for extended error information, which we'll cover in a later chapter. *Connection points* are used to implement callbacks, and again we'll cover these in a later chapter. We'll skip the *free-threaded marshaller* as it's way beyond the scope of this course, but until you are sure of what it means, you probably shouldn't set it.

On clicking OK, the wizard adds some new files to our project, and makes additions to some existing files, as explained on the following slides.



Generated Component-Class

QA-IQ

Account.h

```
// Account.h : Declaration of the CAccount

#include "resource.h"

// CAccount
class ATL_NO_VTABLE CAccount :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CAccount, &CLSID_Account>,
    public IAccount {
public:
    CAccount()
    {}

DECLARE_REGISTRY_RESOURCEID(IDR_ACCOUNT)

BEGIN_COM_MAP(CAccount)
    COM_INTERFACE_ENTRY(IAccount)
END_COM_MAP()

// IAccount
public:
};
```

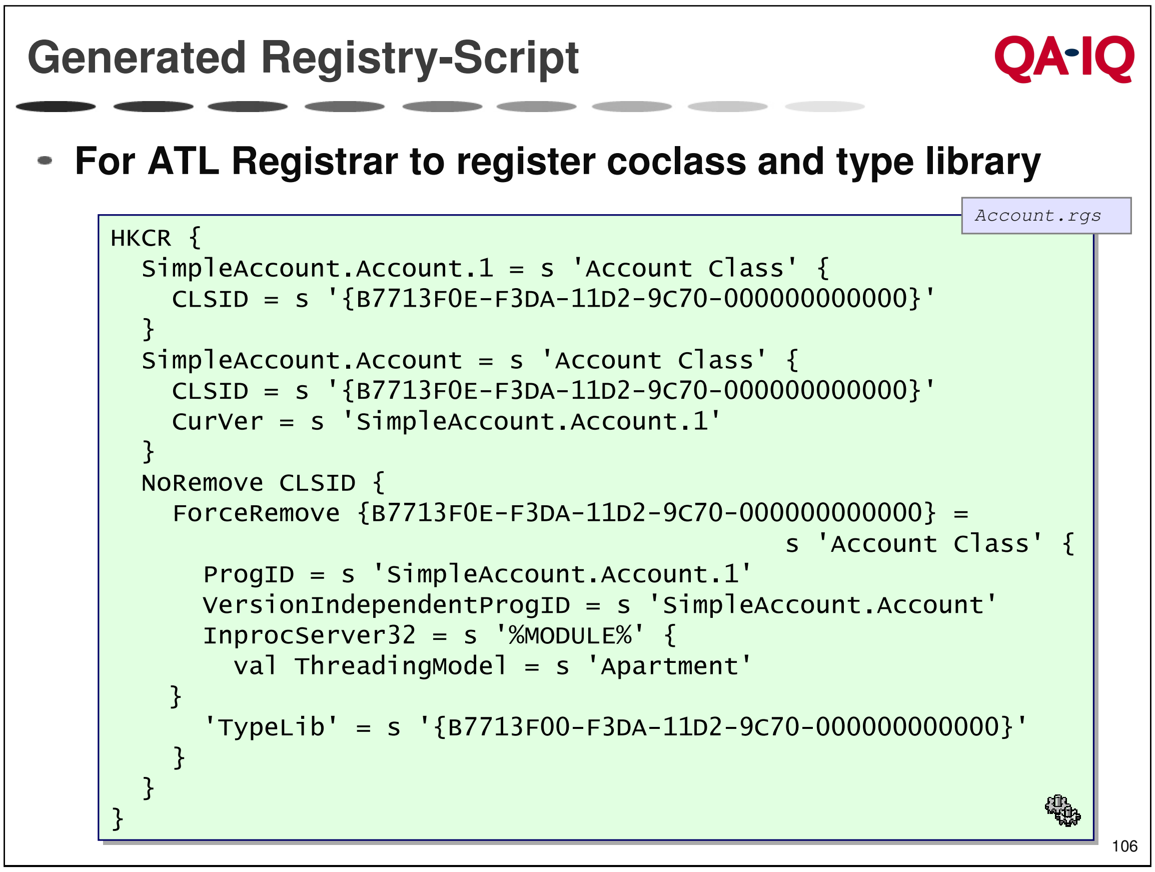
Compiler optimisation to suppress creation of a vtable
Templated classes
Our custom interface
Provides resource ID for registry script
COM interface map for this object
105

The *ATL Object Wizard* adds three new files to our project: `Account.h`, `Account.cpp` and `Account.rgs`.
As shown on the slide, `Account.h` contains the declaration of our Account object's class, `CAccount`. Notice that it multiply inherits from three classes, and two of these are templated classes; we'll discuss these towards the end of the chapter. The third is our custom interface, `IAccount`.
The declaration of `CAccount` contains a macro, `ATL_NO_VTABLE`, which is equivalent to `__declspec(novtable)`. This is a compiler optimisation to suppress the creation of a vtable for this class. At first sight, this seems rather surprising, since we have already learnt that the binary compatibility of a COM object depends on its vtable. However, as we'll discover later, the class of our ATL COM object is never instantiated directly. Instead, it acts a base class to a concrete templated class that can be instantiated.

The next macro, `DECLARE_REGISTRY_RESOURCEID()` simply provides a resource ID for the registry script, `Account.rgs`, which we'll discuss on the next slide.

The two macros, `BEGIN_COM_MAP()` and `END_COM_MAP()` define the *COM (interface) map*. `BEGIN_COM_MAP()` is a complex macro, but basically this includes the definition of an array of `_ATL_INTMAP_ENTRY` elements, where each element associates an interface identifier (IID) with a way of getting a pointer to that interface. The *ATL Object Wizard* has put a single entry in this map using a `COM_INTERFACE_ENTRY()` macro, which means that the only interface exposed by our Account object is `IAccount`. Note that `IUnknown` is not specified here because it is assumed, i.e. all COM objects must expose `IUnknown`, and the first entry in the map is used to provide `IUnknown`.

`Account.cpp` is almost empty, since we've yet to define any methods or properties for the Account object.



Generated Registry-Script

QA-IQ

- **For ATL Registrar to register coclass and type library**

Account.rgs

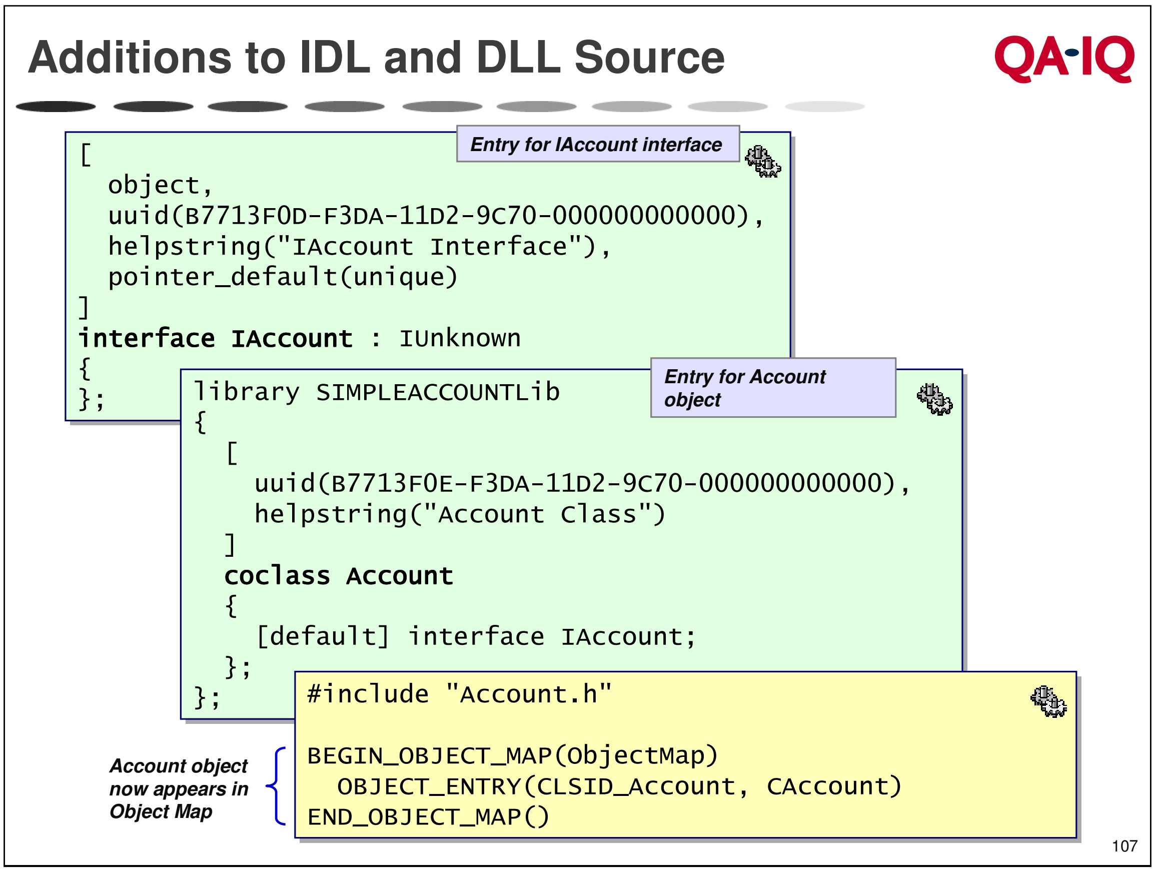
```
HKCR {
  SimpleAccount.Account.1 = s 'Account Class' {
    CLSID = s '{B7713F0E-F3DA-11D2-9C70-000000000000}'
  }
  SimpleAccount.Account = s 'Account Class' {
    CLSID = s '{B7713F0E-F3DA-11D2-9C70-000000000000}'
    CurVer = s 'SimpleAccount.Account.1'
  }
  NoRemove CLSID {
    ForceRemove {B7713F0E-F3DA-11D2-9C70-000000000000} =
                                        s 'Account Class' {
      ProgID = s 'SimpleAccount.Account.1'
      VersionIndependentProgID = s 'SimpleAccount.Account'
      InprocServer32 = s '%MODULE%' {
        val ThreadingModel = s 'Apartment'
    }
      'TypeLib' = s '{B7713F00-F3DA-11D2-9C70-000000000000}'
    }
  }
}
```

106

`Account.rgs` contains a script for registering and unregistering the Account object. This script is compiled into a custom resource and then bound to the server DLL. When the server is requested to register itself (i.e. when `DllRegisterServer()` is called), a *Registrar* component will interpret this resource (i.e the compiled script) and call the appropriate Win32 registry APIs to add the specified entries to the Registry. Similarly, when the server is requested to unregister itself, the Registrar will remove the specified entries from the registry.

An implementation of a Registrar component is supplied with Visual C++ in a DLL called `Atl.dll`, and this component is registered on installation. However, if you want to register our Account object on another machine, you must make sure that `Atl.dll` is present and registered on that machine. Alternatively, a C++ object version of the Registrar can be statically linked to the server code by specifying the *MinDependency* release build option in the project.

As shown on the slide, the syntax of a registry script is reasonably intuitive to a C/C++ programmer. For more information, see the *ATL Registry Component (Registrar)* in the *Visual C++ Documentation*. Alternatively, if you're not familiar with the Backus Nauer Form (BNF) syntax (we're certainly not!), see Appendix B of *ATL COM Programmer's Reference* (Wrox Press).



Additions to IDL and DLL Source

QA-IQ

Entry for IAccount interface

```
[
  object,
  uuid(B7713F0D-F3DA-11D2-9C70-000000000000),
  helpstring("IAccount Interface"),
  pointer_default(unique)
]
interface IAccount : IUnknown
{
};
```

Entry for Account object

```
library SIMPLEACCOUNTLib
{
  [
    uuid(B7713F0E-F3DA-11D2-9C70-000000000000),
    helpstring("Account Class")
  ]
  coclass Account
  {
    [default] interface IAccount;
  };
};
```

```
#include "Account.h"

BEGIN_OBJECT_MAP(ObjectMap)
  OBJECT_ENTRY(CLSID_Account, CAccount)
END_OBJECT_MAP()
```

Account object now appears in Object Map

107

The *ATL Object Wizard* also makes additions to the IDL file (`SimpleAccount.idl`) and the DLL source (`SimpleAccount.cpp`).

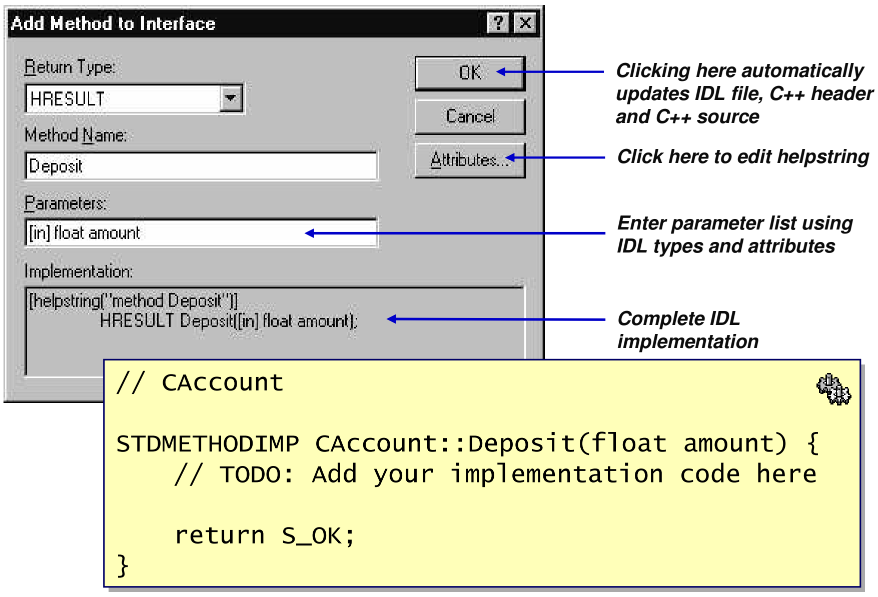
As shown above, the IDL file now contains an `interface` statement for `IAccount`. It also contains a `coclass` statement for the `Account` object. The wizard has automatically generated a couple of new GUIDs, so after running the MIDL compiler, these GUIDs will be defined in `SimpleAccount_i.c` as `IID_IAccount` and `CLSID_Account`, respectively.

In the DLL source, the object map now contains an entry for the Account object. As shown, the `OBJECT_ENTRY` macro takes the CLSID of the object class.

Adding Methods

QA-IQ

- Right-click on IAccount in ClassView, then select Add Method ...



```
[helpstring("method Deposit")]
        HRESULT Deposit([in] float amount);
```

Clicking here automatically updates IDL file, C++ header and C++ source

Click here to edit helpstring

Enter parameter list using IDL types and attributes

Complete IDL implementation

```
// CAccount

STDMETHODIMP CAccount::Deposit(float amount) {
    // TODO: Add your implementation code here

    return S_OK;
}
```

108

We now have a DLL server with a single Account object, but it's not very useful because the component class doesn't have any methods or properties. So, the next step is to define and implement the methods of the `IAccount` interface. Again, this is very easy: we simply select *ClassView*, right click on *IAccount* (either one, it doesn't matter), select *Add Method…*from the popup menu, and fill in the *Add Method to Interface* dialog as shown on the slide.

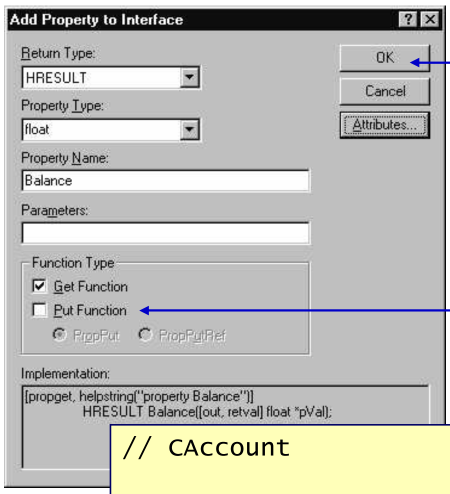
Since we have specified a custom interface, this dialog lets us specify the return type, but as mentioned previously, all COM interface functions should really return an `HRESULT`. After entering the method name, we need to specify its parameter list using IDL types and attributes. The complete IDL implementation is updated as you type. By default, the method gets a `helpstring` that's simply *"method <method name>"*. If necessary, we can change this via an *Edit Attributes* dialog, which is obtained by clicking on the *Attributes* button.

When the *OK* button is clicked, a skeleton method is added to the component C++ source, as shown on the slide.

Adding Properties

QA-IQ

- **Right-click on IAccount in ClassView, then select Add Property**



Return Type: HRESULT
Property Type: float
Property Name: Balance
Parameters:
[✓] Get Function
[ ] Put Function
Implementation:

```
[propget, helpstring("property Balance")]
        HRESULT Balance([out, retval] float *pVal);
```

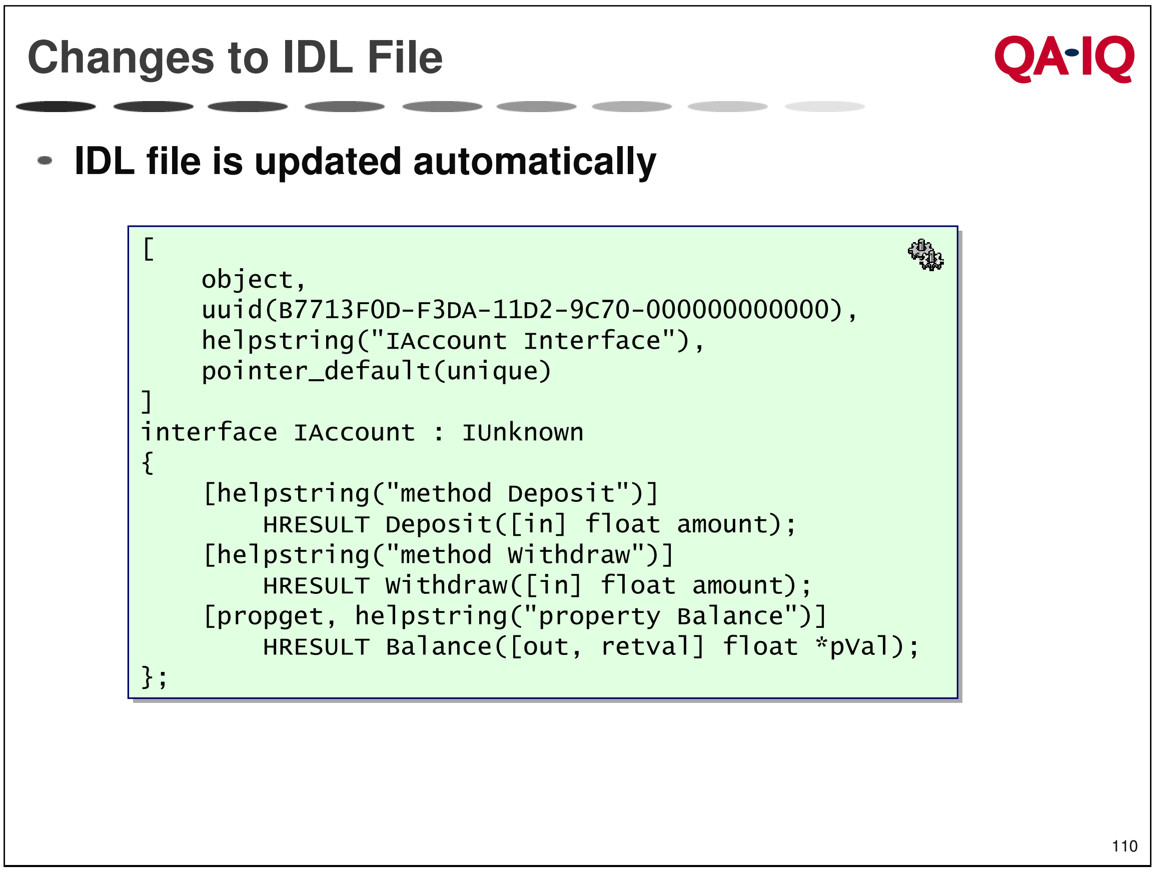
Clicking here automatically updates IDL file, C++ header and C++ source

Deselect Put Function for read-only property

```
// CAccount

STDMETHODIMP CAccount::get_Balance(float *pVal) {
    // TODO: Add your implementation code here

    return S_OK;
}
```

109

Adding a new property to an interface of a COM object is very similar to adding a method. The main difference is that we must specify the function type: *Put* and/or *Get*. Our `IAccount` interface is to have a read-only property called *Balance*, so we've unchecked *Put Function*. So, when we click the *OK* button, a skeleton `get_Balance()` method is added to the component C++ source, as shown on the slide. Note that this method has automatically been given an *out* parameter of the specified property type.



Changes to IDL File

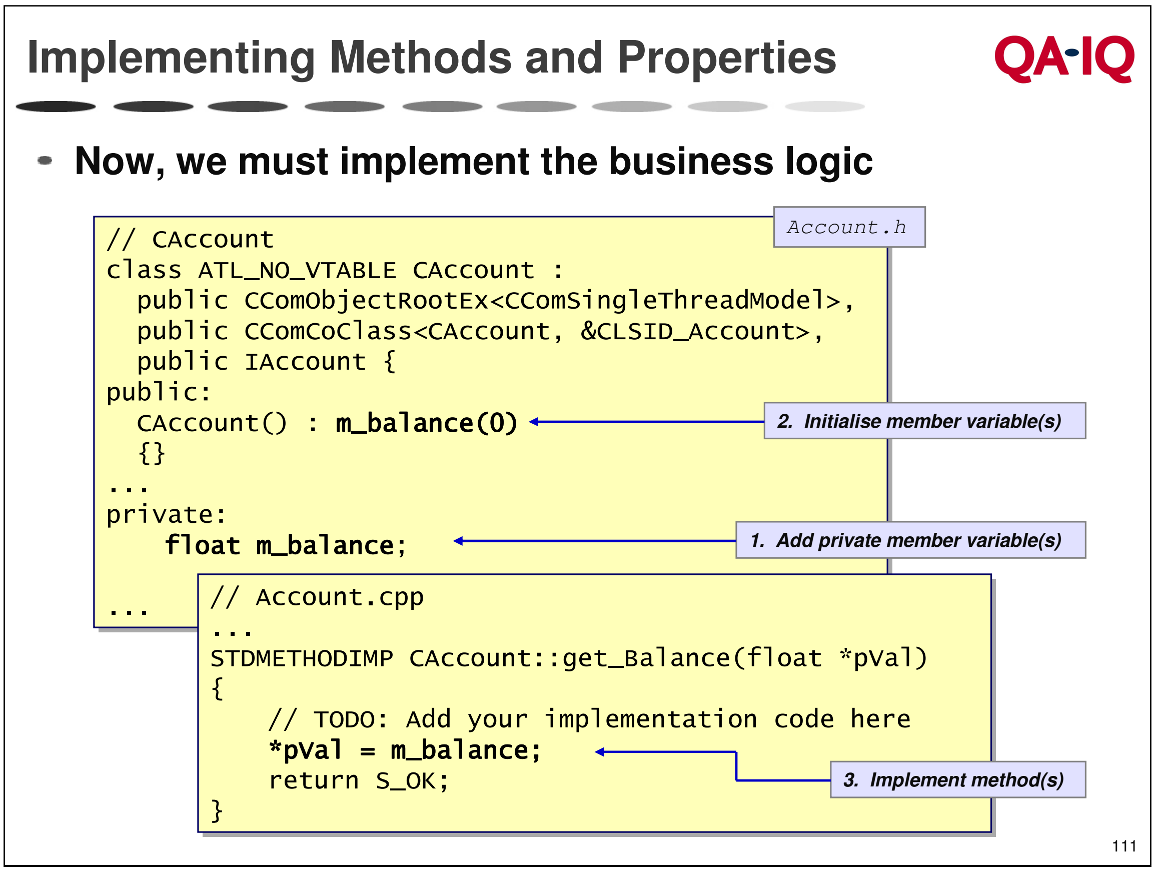
QA-IQ

- **IDL file is updated automatically**

```
[
    object,
    uuid(B7713F0D-F3DA-11D2-9C70-000000000000),
    helpstring("IAccount Interface"),
    pointer_default(unique)
]
interface IAccount : IUnknown
{
    [helpstring("method Deposit")]
        HRESULT Deposit([in] float amount);
    [helpstring("method Withdraw")]
        HRESULT Withdraw([in] float amount);
    [propget, helpstring("property Balance")]
        HRESULT Balance([out, retval] float *pVal);
};
```

110

The IDL file is also updated automatically to reflect the new methods and property that we've added. If necessary, we could edit these at any time.



Implementing Methods and Properties

QA-IQ

- Now, we must implement the business logic

Account.h

```
// CAccount
class ATL_NO_VTABLE CAccount :
  public CComObjectRootEx<CComSingleThreadModel>,
  public CComCoClass<CAccount, &CLSID_Account>,
  public IAccount {
public:
  CAccount() : m_balance(0)
  {}
...
private:
    float m_balance;
...
```

2. Initialise member variable(s)

1. Add private member variable(s)

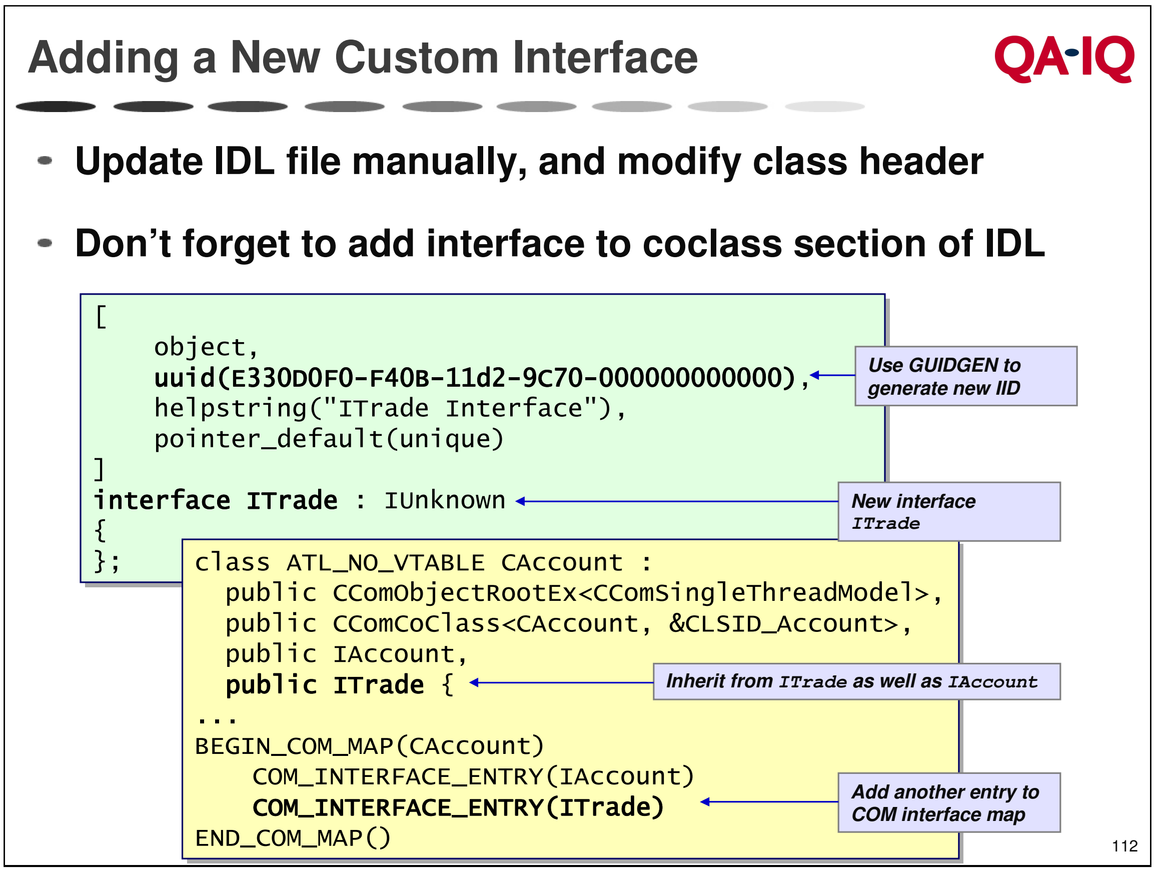
```
// Account.cpp
...
STDMETHODIMP CAccount::get_Balance(float *pVal)
{
    // TODO: Add your implementation code here
    *pVal = m_balance;
    return S_OK;
}
```

3. Implement method(s)

111

Finally, we've got to write some code ourselves to implement the methods and properties of our COM object. In the case of our `Account` object, we need to declare a private member variable, which must be initialised in the constructor.

We could now build and test the server, but before we do, we're going to add another custom interface.



Adding a New Custom Interface

QA-IQ

- Update IDL file manually, and modify class header
- Don't forget to add interface to coclass section of IDL

```
[
    object,
    uuid(E330D0F0-F40B-11d2-9C70-000000000000),
    helpstring("ITrade Interface"),
    pointer_default(unique)
]
interface ITrade : IUnknown
{
};
```

```
class ATL_NO_VTABLE CAccount :
  public CComObjectRootEx<CComSingleThreadModel>,
  public CComCoClass<CAccount, &CLSID_Account>,
  public IAccount,
  public ITrade {
...
BEGIN_COM_MAP(CAccount)
    COM_INTERFACE_ENTRY(IAccount)
    COM_INTERFACE_ENTRY(ITrade)
END_COM_MAP()
```

Use GUIDGEN to generate new IID
New interface ITrade
Inherit from ITrade as well as IAccount
Add another entry to COM interface map

112

Having built our first COM object (with a little help from the ATL wizards!), we'd now like our Account object to implement a new custom interface, `ITrade`. Although Visual C++ provides an *Implement Interface…* wizard (right click on `CAccount` in *ClassView*), this is of limited use because it requires a type library for *ITrade* that we do not have. Therefore, we'll need to update the IDL file and the C++ class header manually.

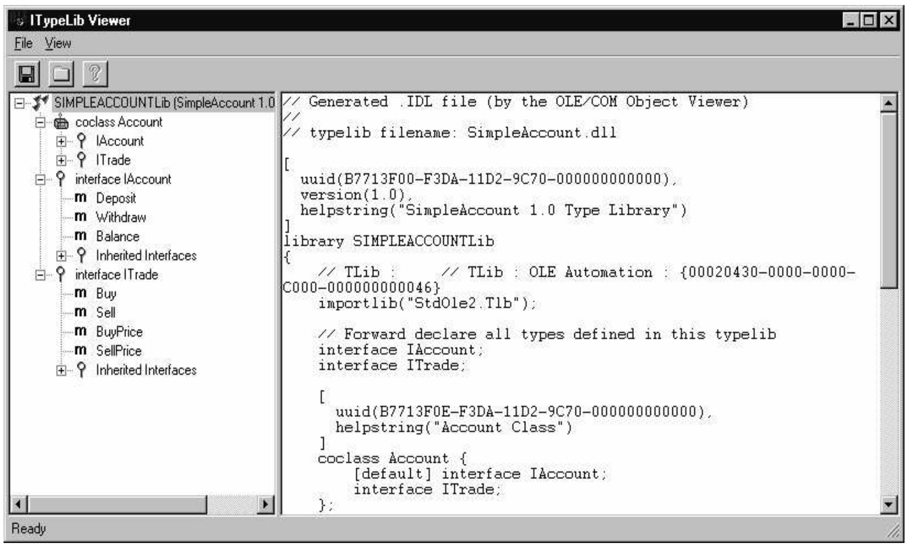
First, we need to use GUIDGEN to generate an IID for the new interface. Then, as shown on the slide, we can insert a new interface statement for `ITrade` in the IDL file, `SimpleAccount.idl`. Next, we need to modify the declaration of `CAccount` in `Account.h`, so that it inherits from `ITrade` in addition to `IAccount`. Also, we mustn't forget to add another entry to the COM map using the `COM_INTERFACE_ENTRY` macro.

Then, it's just a matter of adding methods and properties to `ITrade` in the same way as we did for `IAccount`. Finally, we need to implement those methods and properties.

Building and Testing the Server

QA-IQ

- Build the project
  - Automatically registers the server
  - Check with OLE/COM Object Viewer



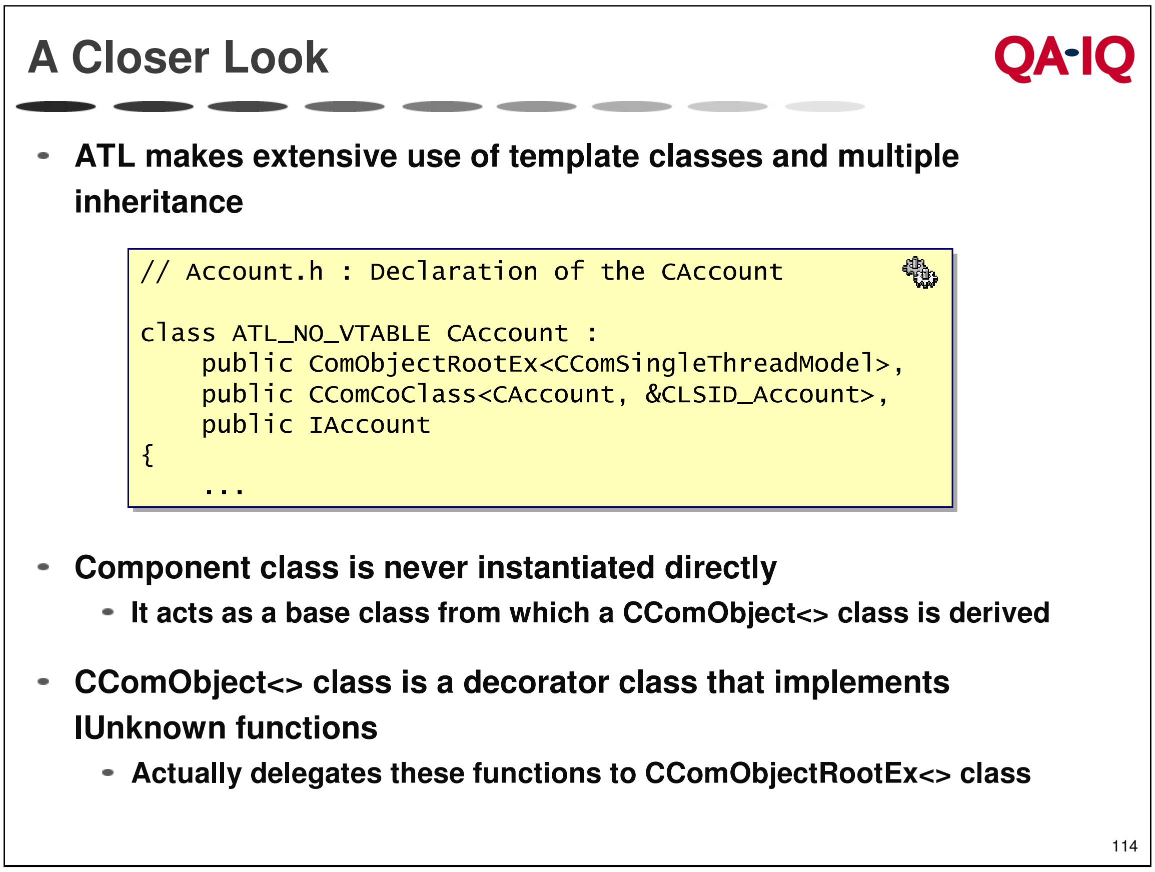
```
// Generated .IDL file (by the OLE/COM Object Viewer)
//
// typelib filename: SimpleAccount.dll

[
  uuid(B7713F00-F3DA-11D2-9C70-000000000000),
  version(1.0),
  helpstring("SimpleAccount 1.0 Type Library")
]
library SIMPLEACCOUNTLib
{
    // TLib :     // TLib : OLE Automation : {00020430-0000-0000-
C000-000000000046}
    importlib("StdOle2.Tlb");

    // Forward declare all types defined in this typelib
    interface IAccount;
    interface ITrade;

    [
      uuid(B7713F0E-F3DA-11D2-9C70-000000000000),
      helpstring("Account Class")
    ]
    coclass Account {
        [default] interface IAccount;
        interface ITrade;
    };
```

113

The project makefile automatically registers the DLL server after a successful build. We can check this with the *OLE/COM Object Viewer*.

After expanding the *All Objects* folder in the left-hand pane, we should find an entry for *Account Class*. Clicking on this entry causes the registry information to be displayed in the right-hand pane. Expanding the folder actually instantiates the Account object so that it can be queried (by calling `QueryInterface()`) to see which interfaces it supports. In this particular case, `IUnknown` is displayed, but there's no sign of `IAccount` or `ITrade`. On reflection, this is hardly surprising since there is nothing in the registry script to register these interfaces; unless we want to provide a proxy-stub DLL, it's not necessary.

To prove that the Account object really does implement the `IAccount` and `ITrade`, we can use the *OLE/COM Object Viewer* to inspect its type library. If we expand the *Type Libraries* folder in the left-hand pane, we should find an entry for *SimpleAccount 1.0 Type Library*. Clicking on this entry causes the type-library information to be displayed in the right-hand pane. Double-clicking on this entry brings up the *ITypeLib Viewer* as shown on the slide. Notice that the right-hand pane shows the IDL that the *OLE/COM Object Viewer* was generated by "de-compiling" the type library.



A Closer Look

- **ATL makes extensive use of template classes and multiple inheritance**

```
// Account.h : Declaration of the CAccount

class ATL_NO_VTABLE CAccount :
    public ComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CAccount, &CLSID_Account>,
    public IAccount
{
    ...
```

- **Component class is never instantiated directly**
  - **It acts as a base class from which a CComObject<> class is derived**
- **CComObject<> class is a decorator class that implements IUnknown functions**
  - **Actually delegates these functions to CComObjectRootEx<> class**

114

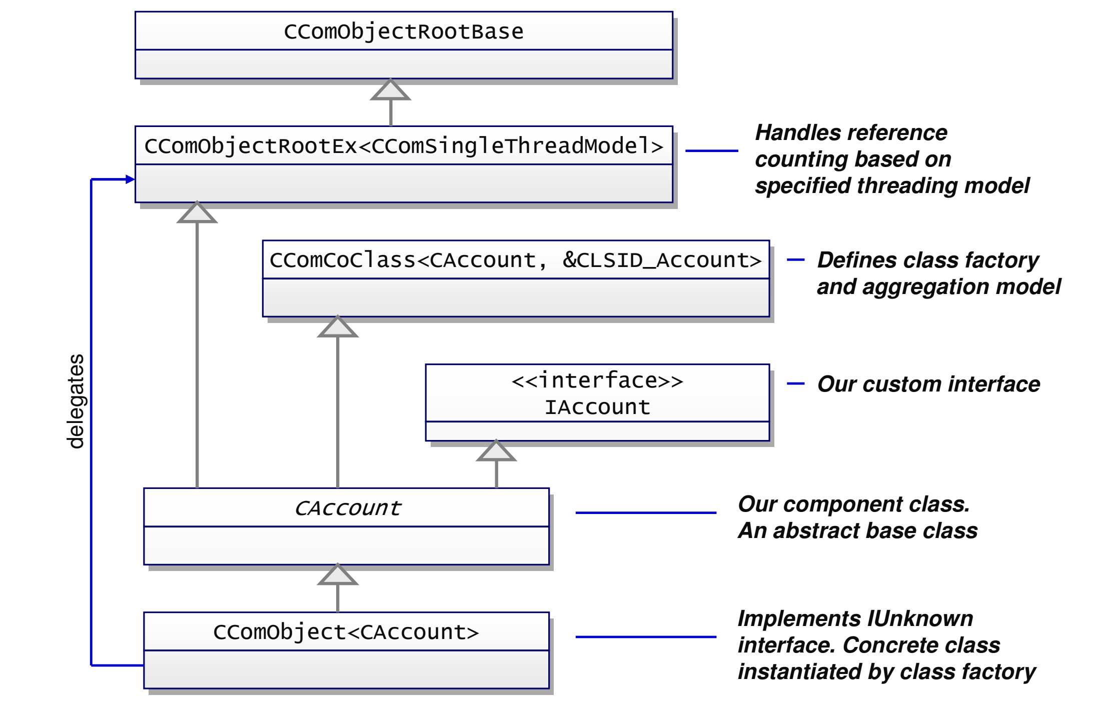
Earlier in this chapter, we promised to explain the two templated classes from which our component class, `CAccount`, derives.

As indicated in the class diagram on the next slide, `CComObjectRootEx<>` handles reference counting based on the specified threading model, which in our case is `CComSingleThreadModel`. `CComSingleThreadModel` provide methods for incrementing and decrementing a value, which for performance reasons are not thread-safe.

`CComCoClass<>` defines the default class factory and aggregation model for the specified class. It also provides methods for obtaining an object's CLSID.

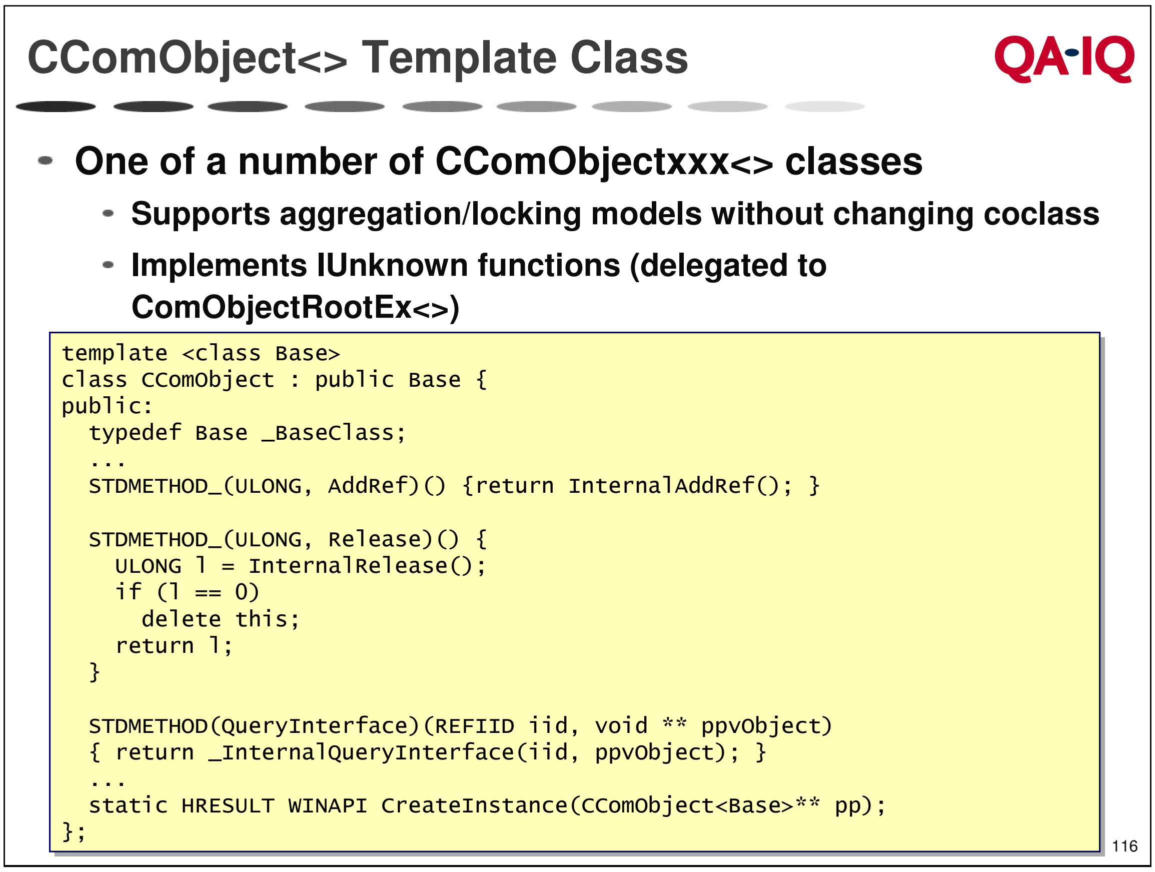
The most surprising thing about the class hierarchy of our component class, `CAccount`, is that it is *not* the most-derived class. If you think about it, this is not really surprising because nowhere in the implementation of `CAccount` is there an implementation of the methods of `IUnknown` (from which `IAccount` derives). Since, in previous chapters, we've learnt that the `IUnknown` methods must be implemented in the most-derived class (to ensure the correct layout of the object's vtable), there must be a more-derived class that does implement the `IUnknown` methods and can therefore be instantiated. This class is yet another templated class, `CComObject<CAccount>`. We'll take a look at this class in a moment.

Class Relationships


CComObjectRootBase
CComObjectRootEx<CComSingleThreadModel>
Handles reference counting based on specified threading model
CComCoClass<CAccount, &CLSID_Account>
Defines class factory and aggregation model
<<interface>> IAccount
Our custom interface
CAccount
Our component class. An abstract base class
CComObject<CAccount>
Implements IUnknown interface. Concrete class instantiated by class factory
delegates

115

The class diagram above shows some of the key relationships between the classes that we've discussed. As mentioned on the previous slide, although `CComObject<>` implements the methods of `IUnknown`, these are delegated to `CComObjectRootEx<>`, a base class of our component class, `CAccount`. What's more, this is completely transparent to `CAccount`.



CComObject<> Template Class

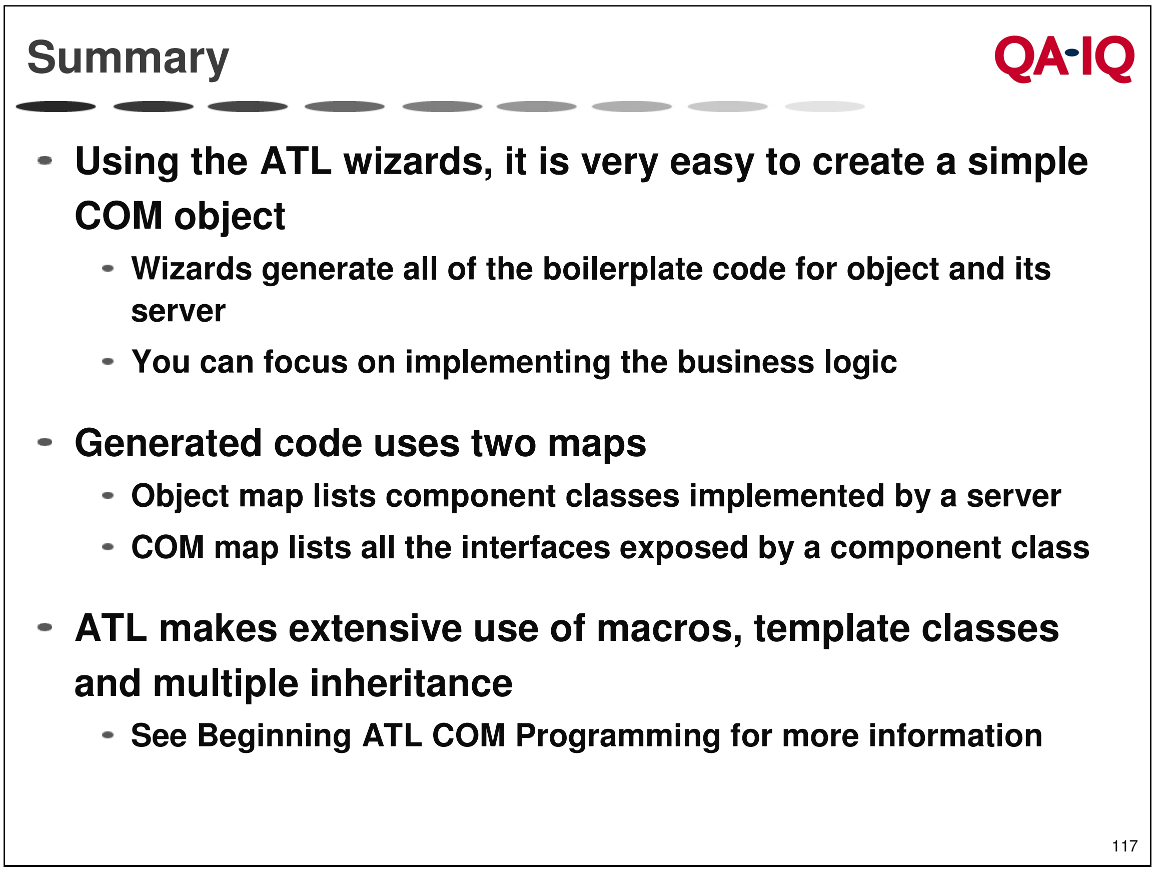
- **One of a number of CComObjectxxx<> classes**
  - **Supports aggregation/locking models without changing coclass**
  - **Implements IUnknown functions (delegated to ComObjectRootEx<>)**

```
template <class Base>
class CComObject : public Base {
public:
  typedef Base _BaseClass;
  ...
  STDMETHOD_(ULONG, AddRef)() {return InternalAddRef(); }

  STDMETHOD_(ULONG, Release)() {
    ULONG l = InternalRelease();
    if (l == 0)
      delete this;
    return l;
  }

  STDMETHOD(QueryInterface)(REFIID iid, void ** ppvObject)
  { return _InternalQueryInterface(iid, ppvObject); }
  ...
  static HRESULT WINAPI CreateInstance(CComObject<Base>** pp);
};
```

116

`CComObject<>` is just one of a number of `CComObjectxxx<>` template classes. Each one implements `IUnknown` according to some predefined rules on whether the class supports aggregation, whether the server needs to be locked when active, whether the object is allocated on the heap or on the stack, etc. This architecture leads to great flexibility, because our component class can be decorated by different decorator classes without requiring any change to our class.

The slide shows some of the implementation of the `CComObject<>` class and clearly shows how the methods of `IUnknown` are delegated to `InternalXXX()` methods, which are methods of `CComObjectRootEx<>`. Note that `CComObject<>` also provides a `CreateInstance()` method.



Summary

QA-IQ

- **Using the ATL wizards, it is very easy to create a simple COM object**
  - **Wizards generate all of the boilerplate code for object and its server**
  - **You can focus on implementing the business logic**
- **Generated code uses two maps**
  - **Object map lists component classes implemented by a server**
  - **COM map lists all the interfaces exposed by a component class**
- **ATL makes extensive use of macros, template classes and multiple inheritance**
  - **See Beginning ATL COM Programming for more information**

117

Useful References:

Grimes et al, *Beginning ATL COM Programming*, Wrox Press, ISBN 1-861000-11-1

Grimes, *ATL COM Programmer's Reference*, Wrox Press, ISBN 1-86100-249-1

Sells and Rector, *ATL Internals*, Addison Wesley, ISBN 0-201-69589-8