/////////////////////////////////////////////////

//

// Smart Interface Class

//

// Copyright 1994-1997, Don Box

//

// There is one class in this file:

//

// SmartInterface<T, PIID> : the smart interface pointer

//

// There is one function in this file:

//

// IsSameObject(x, y) : identity test that supports SmartInterface

//

// There are three macros in this file:

//

// SI(IInterfaceType) : expands to SmartInterface<IInterfaceType, &IID\_IInterfaceType>

// DECLARE\_SMART\_INTERFACE(InterfaceType) : expands to typedef SI(I##InterfaceType) Smart##InterfaceType

// IID\_PPV(sp) : expands to sp.GetIID(), sp.GetAsPPV()

//

// There is one conditional compilation symbol:

// \_NO\_SILENT\_QI : if this symbol is defined, then heterogeneous assignment/construction is disallowed

//

// Usage:

/\*

DECLARE\_SMART\_INTERFACE(PersistStorage)

IPersistStorage \*LoadFromStorage(IStorage \*pstg)

{

CLSID clsid;

ReadClassStg(pstg, &clsid);

SmartInterface<IUnknown, &IID\_IUnknown> punk1;

SI(IUnknown) punk2;

SmartPersistStorage pPersistStg1, pPersistStg2;

// use the IID\_PPV macro to get the last two args in QI form

CoCreateInstance(clsid, 0, CLSCTX\_ALL, IID\_PPV(punk1));

// I1 <- I1 assignment simply addrefs

punk2 = punk1;

// I2 <- I1 assignment silently QI's

pPersistStg1 = punk1;

// I2 <- I2 simply addref's

pPersistStg2 = pPersistStg2;

// I1 <- I2 simply addref's (down cast)

punk2 = pPersistStg2;

// overloaded arrow allows all members EXCEPT for AddRef/Release

pPersistStg2->Load(pstg);

// must explictitly specify usage, since

// dtors will release all outstanding references

// (silent typecast operator allowed too many errors)

CoLockObjectExternal(punk1.GetNonAddRefedInterface(), TRUE, TRUE);

return pPersistStg1.GetAddRefedInterface();

}

\*/

//

//

#ifndef \_SMARTIF\_H

#define \_SMARTIF\_H

#include <assert.h>

struct SmartToken { IUnknown \*m\_pif; SmartToken(IUnknown\* p) : m\_pif(p) {} };

// this routine should really be moved to a real extern

inline BOOL IsSameObject(IUnknown \*pUnk1, IUnknown \*pUnk2)

{

assert(pUnk1 && pUnk2);

BOOL bResult = TRUE;

if (pUnk1 != pUnk2)

{

HRESULT hr;

IUnknown \*p1 = 0, \*p2 = 0;

hr = pUnk1->QueryInterface(IID\_IUnknown, (void\*\*)&p1);

if (SUCCEEDED(hr)) p1->Release(); else p1 = 0;

hr = pUnk2->QueryInterface(IID\_IUnknown, (void\*\*)&p2);

if (SUCCEEDED(hr)) p2->Release(); else p2 = 0;

bResult = (p1 == p2); // values do not change after release

}

return bResult;

}

inline BOOL IsSameObject(IUnknown \*pUnk1, const SmartToken& pUnk2)

{

return IsSameObject(pUnk1, pUnk2.m\_pif);

}

inline BOOL IsSameObject(const SmartToken& pUnk1, IUnknown \*pUnk2)

{

return IsSameObject(pUnk1.m\_pif, pUnk2);

}

inline BOOL IsSameObject(const SmartToken& pUnk1, const SmartToken& pUnk2)

{

return IsSameObject(pUnk1.m\_pif, pUnk2.m\_pif);

}

template <class If, const IID \*piid>

class SmartInterface

{

If \*m\_pif;

typedef SmartInterface<If, piid> SameSmartType;

void SafeRelease(void)

{

if (m\_pif)

m\_pif->Release();

}

public:

// constructors/destructors ///////////////////////////

// default constructor

SmartInterface(void)

: m\_pif(0)

{

}

// homogeneous raw constructor

SmartInterface(If \*rhs)

{

if (m\_pif = rhs)

m\_pif->AddRef();

}

// homogeneous smart constructor

SmartInterface(const SameSmartType& rhs)

{

if (m\_pif = rhs.m\_pif)

m\_pif->AddRef();

}

operator SmartToken (void) const

{

return SmartToken(m\_pif);

}

#if !defined(\_NO\_SILENT\_QI)

// heterogeneous raw constructor

SmartInterface(IUnknown \* rhs)

: m\_pif(0)

{

if (rhs)

rhs->QueryInterface(GetIID(), (void \*\*)&m\_pif);

}

// heterogeneous smart constructor

SmartInterface(const SmartToken& rhs)

: m\_pif(0)

{

if (rhs.m\_pif)

rhs.m\_pif->QueryInterface(GetIID(), (void \*\*)&m\_pif);

}

#endif

// destructor

~SmartInterface(void)

{

SafeRelease();

}

// Attach/Detach operations

// homogeneous raw attachment

void Attach(If \* rhs)

{

if (rhs != m\_pif)

{

SafeRelease();

if (m\_pif = rhs)

m\_pif->AddRef();

}

}

// homogeneous smart attachment

void Attach(const SameSmartType& rhs)

{

Attach(rhs.m\_pif);

}

// heterogeneous raw attachment

void Attach(IUnknown \* rhs)

{

SafeRelease();

if (rhs)

rhs->QueryInterface(GetIID(), (void \*\*)&m\_pif);

else

m\_pif = 0;

}

// heterogeneous smart attachment

void Attach(const SmartToken& rhs)

{

Attach(rhs.m\_pif);

}

void Detach(void)

{

SafeRelease();

m\_pif = 0;

}

// assignment operators ////////////////////////

// homogeneous raw assignment

SameSmartType& operator = (If \*rhs)

{

Attach(rhs);

return \*this;

}

// homogeneous smart assignment

SameSmartType& operator = (const SameSmartType& rhs)

{

Attach(rhs);

return \*this;

}

#if !defined(\_NO\_SILENT\_QI)

// heterogeneous raw assignment

SameSmartType& operator = (IUnknown \* rhs)

{

Attach(rhs);

return \*this;

}

// heterogeneous smart assignment

SameSmartType& operator = (const SmartToken& rhs)

{

Attach(rhs);

return \*this;

}

#endif

// equivalence operators (note - no identity tests performed here!)

BOOL operator == (If \* rhs)

{

return m\_pif == rhs;

}

BOOL operator == (const SameSmartType& rhs)

{

return m\_pif == rhs.m\_pif;

}

BOOL operator != (If \*rhs)

{

return m\_pif != rhs;

}

BOOL operator != (const SameSmartType& rhs)

{

return m\_pif != rhs.m\_pif;

}

// CoCreateInstance wrappers

HRESULT CreateInstance(REFCLSID rclsid,

DWORD dwClsCtx = CLSCTX\_ALL,

IUnknown \*pUnkOuter = 0)

{

Detach();

return CoCreateInstance(rclsid, pUnkOuter, dwClsCtx,

GetIID(), AsPPVArg());

}

HRESULT CreateInstance(LPCOLESTR szProgID,

DWORD dwClsCtx = CLSCTX\_ALL,

IUnknown \*pUnkOuter = 0)

{

CLSID clsid;

HRESULT result = CLSIDFromProgID(szProgID, &clsid);

if (SUCCEEDED(result))

result = CreateInstance(clsid, dwClsCtx, pUnkOuter);

return result;

}

HRESULT GetClassObject(REFCLSID rclsid,

DWORD dwClsCtx = CLSCTX\_ALL)

{

Detach();

return CoGetClassObject(rclsid, dwClsCtx, 0,

GetIID(), AsPPVArg());

}

HRESULT GetClassObject(LPCOLESTR szProgID,

DWORD dwClsCtx = CLSCTX\_ALL)

{

CLSID clsid;

HRESULT result = CLSIDFromProgID(szProgID, &clsid);

if (SUCCEEDED(result))

result = GetClassObject(clsid, dwClsCtx);

return result;

}

#ifdef \_WIN32\_DCOM

HRESULT CreateInstance(REFCLSID rclsid, DWORD dwClsCtx,

const OLECHAR \*pwszHostName,

COAUTHINFO \*pai = 0)

{

Detach();

COSERVERINFO csi = { 0, pwszHostName, pai, 0 };

MULTI\_QI mqi = { piid, 0, 0 };

HRESULT hr = CoCreateInstanceEx(rclsid, 0, dwClsCtx,

&csi, 1, &mqi);

if (SUCCEEDED(hr))

\*(IUnknown\*\*)&m\_pif = mqi.pItf;

return hr;

}

HRESULT CreateInstance(LPCOLESTR szProgID, DWORD dwClsCtx,

const OLECHAR \*pwszHostName,

COAUTHINFO \*pai = 0)

{

CLSID clsid;

HRESULT result = CLSIDFromProgID(szProgID, &clsid);

if (SUCCEEDED(result))

result = CreateInstance(clsid, dwClsCtx, pwszHostName, pai);

return result;

}

HRESULT GetClassObject(REFCLSID rclsid,

DWORD dwClsCtx,

const OLECHAR \*pwszHostName,

COAUTHINFO \*pai = 0)

{

Detach();

COSERVERINFO csi = { 0, pwszHostName, pai, 0 };

return CoGetClassObject(rclsid, dwClsCtx, &csi,

GetIID(), AsPPVArg());

}

HRESULT GetClassObject(LPCOLESTR szProgID,

DWORD dwClsCtx,

const OLECHAR \*pwszHostName,

COAUTHINFO \*pai = 0)

{

CLSID clsid;

HRESULT result = CLSIDFromProgID(szProgID, &clsid);

if (SUCCEEDED(result))

result = GetClassObject(clsid, dwClsCtx, pwszHostName, pai);

return result;

}

#endif

HRESULT BindToObject(IMoniker \*pmk, IBindCtx \*pbc = 0, IMoniker \*pmkToLeft = 0)

{

Detach();

HRESULT hr = S\_OK;

if (pbc)

pbc->AddRef();

else

hr = CreateBindCtx(0, &pbc);

if (SUCCEEDED(hr))

{

hr = pmk->BindToObject(pbc, pmkToLeft, GetIID(), (void\*\*)&m\_pif);

if (FAILED(hr))

m\_pif = 0;

pbc->Release();

}

return hr;

}

HRESULT BindToObject(LPCOLESTR pszDisplayName)

{

Detach();

IBindCtx \*pbc = 0;

HRESULT hr = CreateBindCtx(0, &pbc);

if (SUCCEEDED(hr))

{

IMoniker \*pmk = 0;

ULONG cch = 0;

hr = MkParseDisplayName(pbc, pszDisplayName, &cch, &pmk);

if (SUCCEEDED(hr))

{

hr = pmk->BindToObject(pbc, 0, GetIID(), (void\*\*)&m\_pif);

if (FAILED(hr))

m\_pif = 0;

pmk->Release();

}

pbc->Release();

}

return hr;

}

// operations

const IID& GetIID(void) const

{

return \*piid;

}

void \* \* AsPPVArg(void)

{

SafeRelease();

return (void \* FAR\*)&m\_pif;

}

// note: no If \* operator is allowed, as it makes it very

// easy to break the protocol of COM. Instead, these

// four operations require the user to be explicit

If \* GetAddRefedInterface(void) const

{

if (m\_pif)

m\_pif->AddRef();

return m\_pif;

}

If \* GetNonAddRefedInterface(void) const

{

return m\_pif;

}

If \*\*GetReleasedInterfaceReference(void)

{

SafeRelease();

return &m\_pif;

}

If \*\*GetNonReleasedInterfaceReference(void)

{

return &m\_pif;

}

BOOL operator ! (void) const

{

return m\_pif == 0;

}

BOOL IsOK(void) const

{

return m\_pif != 0;

}

// instead of operator bool, we return a fake ptr type to allow if (pFoo) usage

// but to disallow if (pFoo == pBar) which is probably wrong

class PseudoBool {};

operator PseudoBool \* (void) const

{

return (PseudoBool \*)m\_pif;

}

// the arrow operator returns a pointer with AddRef and Release disabled

class NoAddRefOrRelease : public If {

private:

STDMETHOD\_(ULONG, AddRef)(THIS) PURE;

STDMETHOD\_(ULONG, Release)(THIS) PURE;

};

NoAddRefOrRelease \*operator ->(void)

{

assert(m\_pif);

return (NoAddRefOrRelease \*)m\_pif;

}

};

#if \_MSC\_VER>1020

template <>

#endif

class SmartInterface<IUnknown, &IID\_IUnknown>

{

If \*m\_pif;

typedef IUnknown If;

typedef SmartInterface<If, &IID\_IUnknown> SameSmartType;

void SafeRelease(void)

{

if (m\_pif)

m\_pif->Release();

}

public:

// constructors/destructors ///////////////////////////

// default constructor

SmartInterface(void)

: m\_pif(0)

{

}

// homogeneous raw constructor

SmartInterface(If \*rhs)

{

if (m\_pif = rhs)

m\_pif->AddRef();

}

// homogeneous smart constructor

SmartInterface(const SameSmartType& rhs)

{

if (m\_pif = rhs.m\_pif)

m\_pif->AddRef();

}

operator SmartToken (void) const

{

return SmartToken(m\_pif);

}

// heterogeneous raw constructor (see homo version)

// heterogeneous smart constructor (AddRef's instead of QI)

SmartInterface(const SmartToken& rhs)

: m\_pif(0)

{

if (m\_pif = rhs.m\_pif)

m\_pif->AddRef();

}

// destructor

~SmartInterface(void)

{

SafeRelease();

}

// Attach/Detach operations

// homogeneous raw attachment

void Attach(If \* rhs)

{

if (rhs != m\_pif)

{

SafeRelease();

if (m\_pif = rhs)

m\_pif->AddRef();

}

}

// homogeneous smart attachment

void Attach(const SameSmartType& rhs)

{

Attach(rhs.m\_pif);

}

// heterogeneous raw attachment (see homo version)

// heterogeneous smart attachment

void Attach(const SmartToken& rhs)

{

Attach(rhs.m\_pif);

}

void Detach(void)

{

SafeRelease();

m\_pif = 0;

}

// assignment operators ////////////////////////

// homogeneous raw assignment

SameSmartType& operator = (If \*rhs)

{

Attach(rhs);

return \*this;

}

// homogeneous smart assignment

SameSmartType& operator = (const SameSmartType& rhs)

{

Attach(rhs);

return \*this;

}

// heterogeneous raw assignment (see homo version)

// heterogeneous smart assignment

SameSmartType& operator = (const SmartToken& rhs)

{

Attach(rhs);

return \*this;

}

// equivalence operators (note - no identity tests performed here!)

BOOL operator == (If \* rhs)

{

return m\_pif == rhs;

}

BOOL operator == (const SameSmartType& rhs)

{

return m\_pif == rhs.m\_pif;

}

BOOL operator != (If \*rhs)

{

return m\_pif != rhs;

}

BOOL operator != (const SameSmartType& rhs)

{

return m\_pif != rhs.m\_pif;

}

// CoCreateInstance wrappers

HRESULT Instantiate(REFCLSID rclsid,

DWORD dwClsCtx = CLSCTX\_ALL,

IUnknown \*pUnkOuter = 0)

{

Detach();

return CoCreateInstance(rclsid, pUnkOuter, dwClsCtx,

GetIID(), AsPPVArg());

}

HRESULT Instantiate(LPCOLESTR szProgID,

DWORD dwClsCtx = CLSCTX\_ALL,

IUnknown \*pUnkOuter = 0)

{

CLSID clsid;

HRESULT result = CLSIDFromProgID(szProgID, &clsid);

if (SUCCEEDED(result))

result = Instantiate(clsid, dwClsCtx, pUnkOuter);

return result;

}

HRESULT BindToObject(IMoniker \*pmk, IBindCtx \*pbc = 0, IMoniker \*pmkToLeft = 0)

{

Detach();

HRESULT hr = S\_OK;

if (pbc)

pbc->AddRef();

else

hr = CreateBindCtx(0, &pbc);

if (SUCCEEDED(hr))

{

hr = pmk->BindToObject(pbc, pmkToLeft, GetIID(), (void\*\*)&m\_pif);

if (FAILED(hr))

m\_pif = 0;

pbc->Release();

}

return hr;

}

HRESULT BindToObject(LPCOLESTR pszDisplayName)

{

Detach();

IBindCtx \*pbc = 0;

HRESULT hr = CreateBindCtx(0, &pbc);

if (SUCCEEDED(hr))

{

IMoniker \*pmk = 0;

ULONG cch = 0;

hr = MkParseDisplayName(pbc, pszDisplayName, &cch, &pmk);

if (SUCCEEDED(hr))

{

hr = pmk->BindToObject(pbc, 0, GetIID(), (void\*\*)&m\_pif);

if (FAILED(hr))

m\_pif = 0;

pmk->Release();

}

pbc->Release();

}

return hr;

}

// operations

const IID& GetIID(void) const

{

return IID\_IUnknown;

}

void \* \* AsPPVArg(void)

{

SafeRelease();

return (void \* FAR\*)&m\_pif;

}

// note: no If \* operator is allowed, as it makes it very

// easy to break the protocol of COM. Instead, these

// four operations require the user to be explicit

If \* GetAddRefedInterface(void) const

{

if (m\_pif)

m\_pif->AddRef();

return m\_pif;

}

If \* GetNonAddRefedInterface(void) const

{

return m\_pif;

}

If \*\*GetReleasedInterfaceReference(void)

{

SafeRelease();

return &m\_pif;

}

If \*\*GetNonReleasedInterfaceReference(void)

{

return &m\_pif;

}

BOOL operator ! (void) const

{

return m\_pif == 0;

}

BOOL IsOK(void) const

{

return m\_pif != 0;

}

// instead of operator bool, we return a fake ptr type to allow if (pFoo) usage

// but to disallow if (pFoo == pBar) which is probably wrong

class PseudoBool {};

operator PseudoBool \* (void) const

{

return (PseudoBool \*)m\_pif;

}

// the arrow operator returns a pointer with AddRef and Release disabled

class NoAddRefOrRelease : public If {

private:

STDMETHOD\_(ULONG, AddRef)(THIS) PURE;

STDMETHOD\_(ULONG, Release)(THIS) PURE;

};

NoAddRefOrRelease \*operator ->(void)

{

assert(m\_pif);

return (NoAddRefOrRelease \*)m\_pif;

}

};

#define SI(InterfaceName) \

SmartInterface<InterfaceName, &IID\_##InterfaceName>

#define DECLARE\_SMART\_INTERFACE(InterfaceName) \

typedef SI(I##InterfaceName) Smart##InterfaceName

#define IID\_PPV(sif) ((sif).GetIID()), ((sif).AsPPVArg())

#endif