ECOM
EPOC component object model

Background of ECOM

Typically, Symbian OS applications are event based and there are many different kinds of service provider.

Analogous to MS windows COM/DCOM

An physical example

- Client just drop the mail, he may not know how the mail will move,
- Client may tell post office that he sends via air,
- If there were no air line route the system would work, but client can't use it,
- If another system come dynamically it can just work and client can use it
- Client does not know how post office handle this (route the mail)

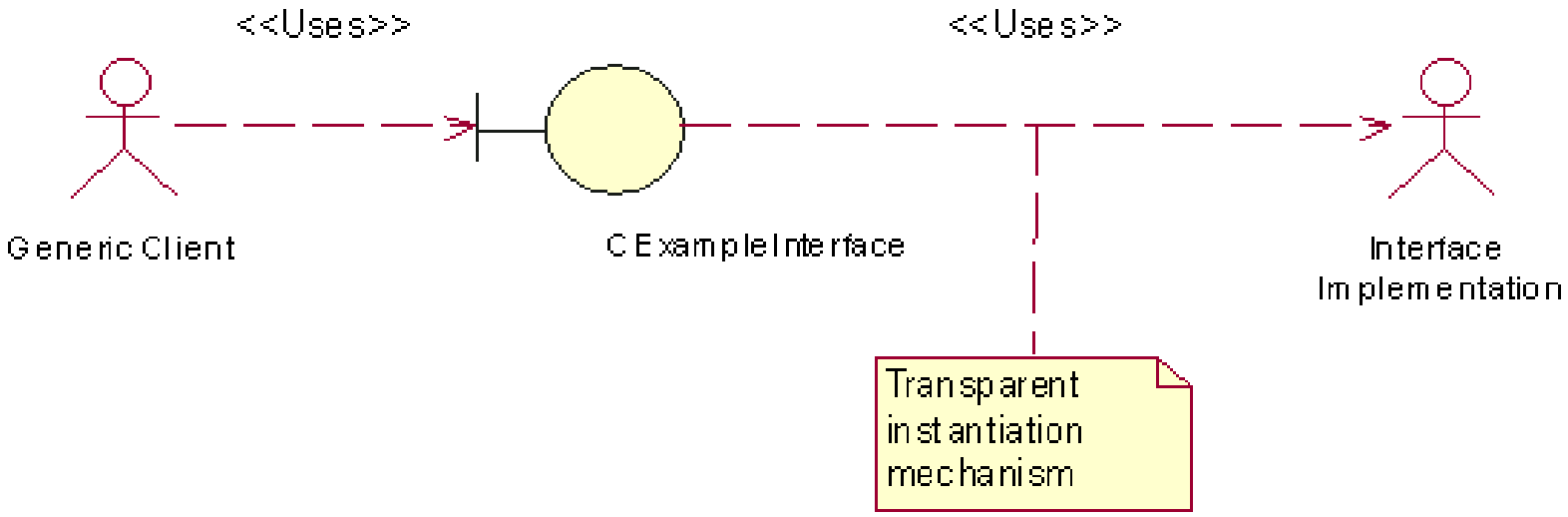
Background of ECOM cont'

- In the previous system, client can be thought as client application that provide service to end users or any other software,
- Post office can be thought as a ECOM framework and back bone of the system
- Actual mail delivery via air, land or ship can be thought as an ECOM plug-ins that does the actual work.
- In symbian there are lots of plug-ins:
- For example followings (or its components) are built with ECOM plug-in
  - Message sending via SMS
  - Message sending via MMS
  - Message sending via BT
  - Message sending via IR and so on
- Advantages of ECOM plug-ins
  - Early (pre 7.0) Symbian OS has polymorphic interface DLL, all clients who needs to use a DLL need to provide their own mechanism for client to discover and instantiate avaiable implementations.
  - ECOM resolve this duplication by providing generic framework
    - Register and discover interface implementations,
    - Select an appropriate implementation to use,
    - Provide version control for plug-ins,

Background of ECOM cont’

- A client wishes to access an object to perform some processing.
- The specifics of this object are not known until run-time.
- The general characteristics of the processing are known, and are defined in an interface, but several variants of required processing could exist, which are provided by implementations that support the interface
- There are four clearly-defined roles in such a system.
  - the client that wishes to access services,
  - the interface that defines how to request services,
  - the interface implementation that provides the required processing,
  - the framework that provides clients with access to the implementations.

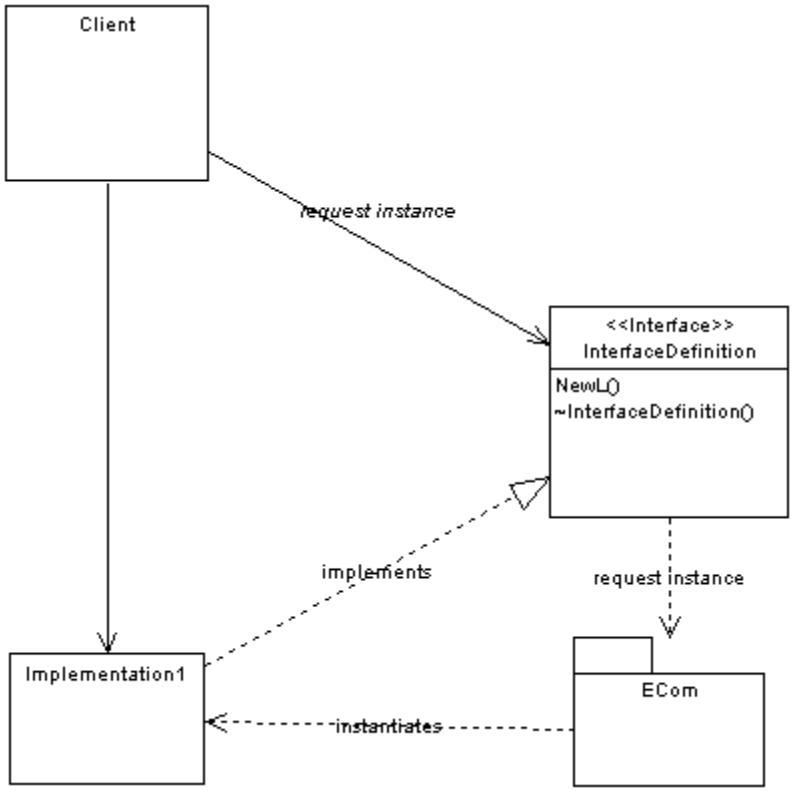
Background of ECOM cont’



- Client create an instance via interface
- Client uses the instance
- Client call method on the instance
- Client delete the instance

Interface definitions

- Interface definition serves two purposes:
  - it defines functions that offer services to clients. These functions are typically pure virtual. Implementations derive from the interface to provide concrete functionality.
  - it offers object instantiation functions. Such a function does *not* create an instance of the interface definition class itself: instead, it uses ECom to create an instance of an appropriate implementation, and returns that to the client. The interface's destructor similarly notifies ECom that the object is destroyed.



Interface definitions

- Interface definition provides to ECom an indication of which of the available implementations should be used. This indication can be very explicit, such as a UID that identifies a particular implementation, or indirect, such as data to match against an implementation's self-description.
- This selection process is called *resolution*, and the entity that performs it a *resolver*. ECom provides a default resolver, but interface definitions can provide their own specialised resolvers where required

Resolution

- The framework supplies a default resolver for selecting appropriate implementations. This attempts to match data supplied by the client against a data identifier field in the implementation's registration details. The interface can ask the framework to create the best fit implementation directly, or ask it to return a list of all matching implementations, and then possibly use other methods, such as user selection, to select the one to be created.

Interface definitions

```
class CExampleInterface : public CBase
    {
public:
    // Instantiates an object of this type
    static CExampleInterface* NewL();

    // Instantiates an object of this type using the aMatchString as the
    resolver parameters.
    static CExampleInterface* NewL(const TDesC8& aMatchString);
    // Destructor.
    virtual ~CExampleInterface();
    // Request a list of all available implementations which
    // satisfy this given interface.
    static void ListAllImplementationsL(RImplInfoPtrArray& aImplInfoArray);
    // Pure interface method Representative of a method provided on the
    interface by
    // the interface definer.
    virtual void DoMethodL(TDes& aString) = 0;

protected:
    //Default c'tor
    inline CExampleInterface();

private:
    // Unique instance identifier key
    TUid iDtor_ID_Key;
    };
```

Resource Registration

To write a standard registration resource file:

- Set the file name to be the UID of the DLL (the same as the second number in the project file's UID statement, and as the value of the dll_uid member below) in hexadecimal, for example, 10009DB0.rss.
- The registration resource structure types are defined RegistryInfo.rh, so include that file define a single REGISTRY_INFO resource. Set its dll_uid member to be the DLL's UID. Set its interfaces member to be an array of INTERFACE_INFO resources. each INTERFACE_INFO resource defines registration information for implementations of the interface identified by its interface_uid member.

- The information takes the form of an array of IMPLEMENTATION_INFO resources each IMPLEMENTATION_INFO resource declares the properties of a single implementation. It has five members:
  - implementation_uid: the unique identifier for this implementation
  - version_no: the version of this interface implementation
  - display_name: the implementation's human-readable name
  - default_data: the data identifier field which a resolver uses to determine if the implementation matches a client request
  - opaque_data: a binary data field, unused by ECom, which can contain additional data, for example, for use by custom resolvers

Resource Registration

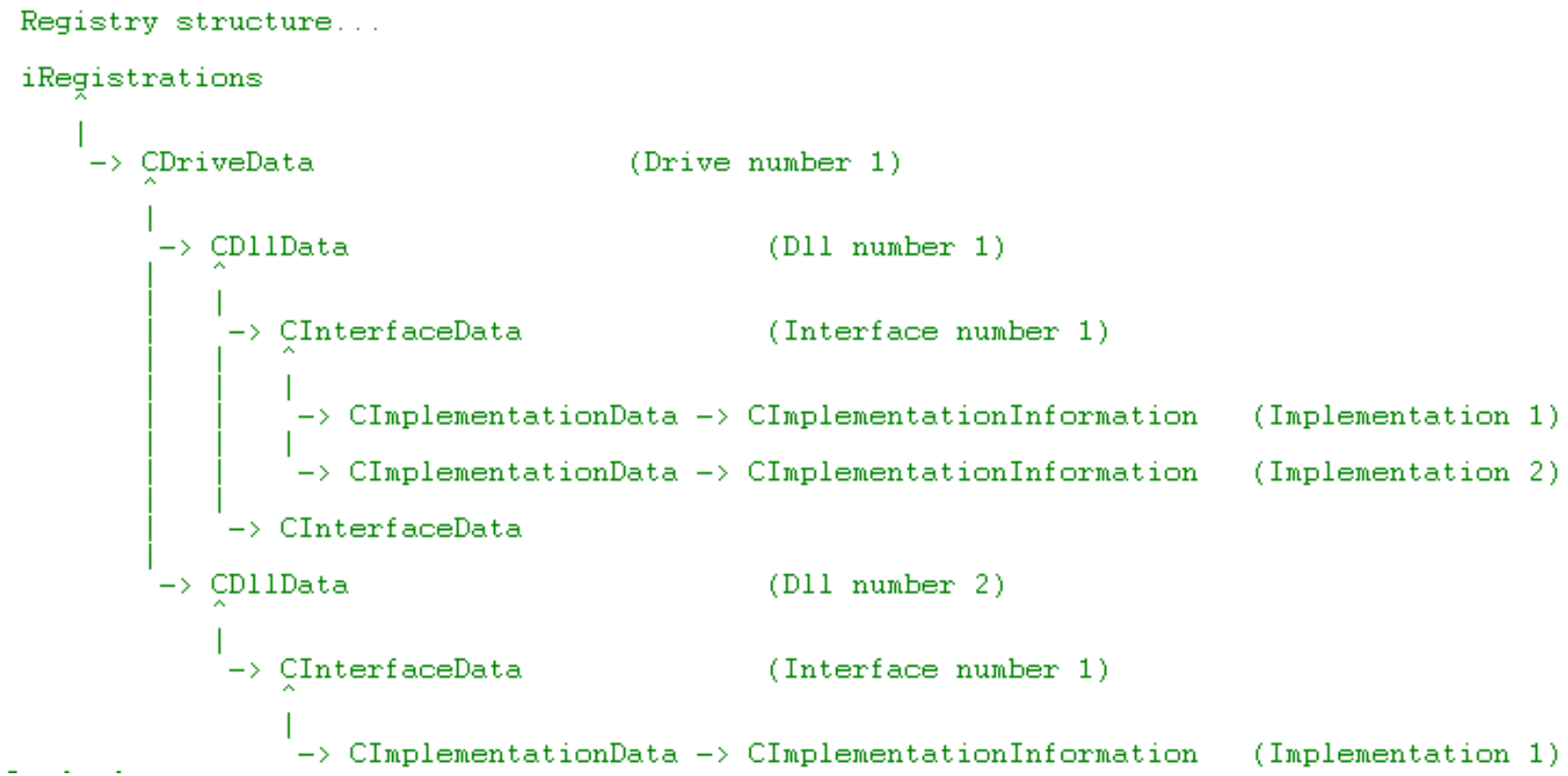
// RegistryInfo.rh

```
STRUCT REGISTRY_INFO
  {
  LONG dll_uid;
  STRUCT interfaces[];
  }

STRUCT INTERFACE_INFO
  {
  LONG interface_uid;
  STRUCT implementations[];
  }


STRUCT IMPLEMENTATION_INFO
  {
  LONG implementation_uid;
  BYTE version_no;
  LTEXT8 name;
  LTEXT8 data_type;
  LTEXT8 match_params;
  }
```

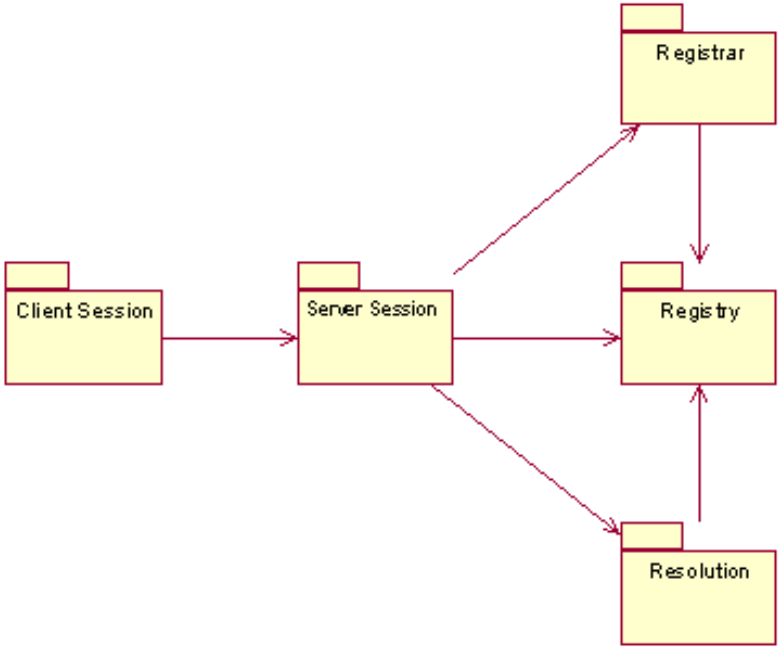
```
// 10009DB1.RSS
#include "RegistryInfo.rh"
RESOURCE REGISTRY_INFO theInfo
{
dll_uid = 0x10009DB1;
interfaces =
    {
    INTERFACE_INFO
        {
        interface_uid = 0x10009DC0;
        implementations =
            {
            IMPLEMENTATION_INFO
                {
                implementation_uid = 0x10009DC3;
                version_no = 1;
                display_name = "Implementation name 1";
                default_data = "text/wml";
                opaque_data = "some extra data";
                },
            IMPLEMENTATION_INFO
                {
                implementation_uid = 0x10009DC4;
                version_no = 1;
                display_name = "Implementation name 1";
                default_data = "text/xml||Type of data handled"
                opaque_data = "";
                }
            };
        }
    };
}
```

Resource Registration

Registry data in a derive



Resource Registration



•To provide efficient and swift resolution with a minimal footprint, the framework employs the standard client server architecture.

•The client side provides a session class REComSession with a rich interface tailored to creating, and destroying implementations

- •Addition, and removal of interface implementations. (CRegisrar functions).
- •Access, and persistence mechanisms. (CRegisryData functions)
- •Resolution, and searching mechanisms. (Cresolver functions)
- •Load, and unload control. (CloadManager functions)

•This last sub-system has rather a simple set of functionality, and so for implementation purposes has been combined with the Resolution sub-system, upon which it depends.

Interface implementations

- An *interface implementation* provides the services promised by the interface. An implementation does this in the standard C++ way by deriving from the base class, in this case, the interface definition, and implementing its functions.
- One or more interface implementations are gathered together in a DLL, termed an *interface implementation collection*. A collection can have implementations of more than one interface. Aside from the implementations themselves, each collection has two key features that allow ECom to use it:
  - it exports an array of factory functions that allow implementations to be created
  - it publishes a compiled resource file that lists its implementations and their properties

Export implementation factories

- An interface implementation collection gathers one or more interface implementations in a DLL and provides necessary information for ECom to use them.

- The collection must export a single function that provides an array that maps a UID for each implementation to its respective creation function

- The signature of this exported function is:

```
EXPORT_C const TImplementationProxy* ImplementationGroupProxy(TInt& aTableCount);

typedef struct
{
TUid      iImplementationUid;
TProxyNewLPtr iNewLFuncPtr;
} TImplementationProxy;
```

Export implementation factories

- ECOM frame work calls ImplementationGroupProxy function to get a pointer of `TImplementationProxy`

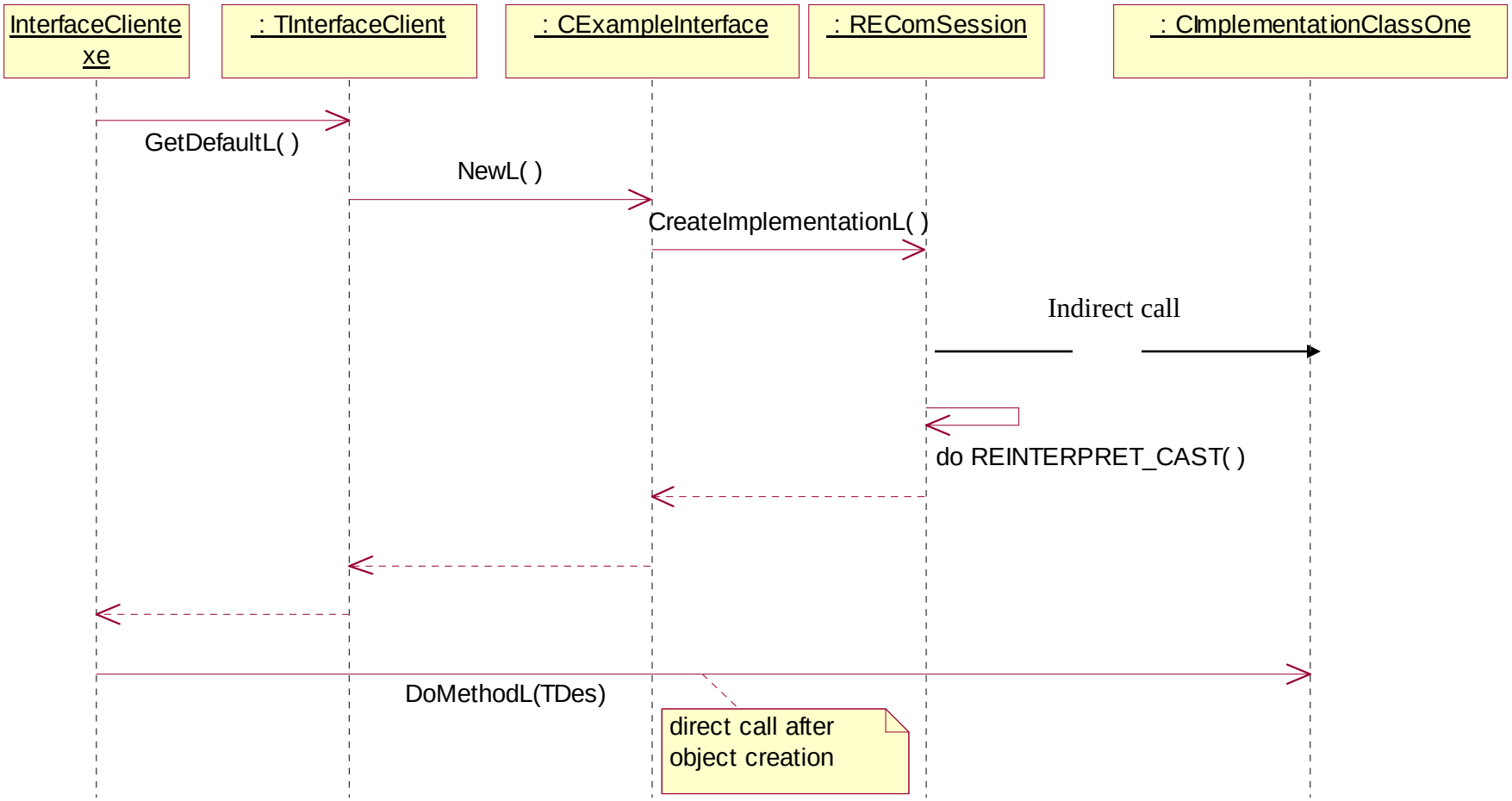
```
// Define the interface UIDs
    const TImplementationProxy ImplementationTable[] =
        {
        {{0x10009DC3}, CImplementationClassOne::NewL},
        {{0x10009DC4}, CImplementationClassTwo::NewL}
        };

    EXPORT_C const TImplementationProxy* ImplementationGroupProxy(TInt&
    aTableCount)
        {
        aTableCount = sizeof(ImplementationTable) /
    sizeof(TImplementationProxy);

        return ImplementationTable;
        }
```

- The UID in the array must be the same as the implementation_uid for the implementation in its registration resource file.
- Normally implemented in `Proxy.cpp` file

MMP File for Plug-in

```
// ExampleInterfaceImplementation.mmp
    TARGET ExampleInterfaceImplementation.dll
    TARGETTYPE PLUGIN
    // ECom Dll recognition UID followed by the unique dll UID
    UID 0x10009D8D 0x10009DB0
//resource file name 10009DB0.rss and dll_uid in 10009DB0.rss should be 0x10009DB0
    VENDORID 0
CAPABILITY NONE
    SOURCEPATH \ExampleInterfaceImplementation
    SOURCE ExampleInterfaceImplementation.cpp

    USERINCLUDE \ExampleInterfaceImplementation \inc
    SYSTEMINCLUDE \epoc32\include
    SYSTEMINCLUDE \epoc32\include\ecom

    START RESOURCE
    10009DB0.rss
    TARGET          ExampleInterfaceImplementation.rsc
    END
    LIBRARY euser.lib
    LIBRARY ECom.lib // ecom static library
```

Sequence


InterfaceClientexe
: TInterfaceClient
: CExampleInterface
: REComSession
: CImplementationClassOne
GetDefaultL( )
NewL( )
CreateImplementationL( )
Indirect call
do REINTERPRET_CAST( )
DoMethodL(TDes)
direct call after object creation