Roadmap to COM

Jim Fawcett

CSE775 - Distributed Objects

Spring 2014

Roadmap

- What's the problem?
  - tight coupling between many components in large systems makes debugging, integration, and maintenance very difficult
  - builds are complex activities depending on many different pieces of source code and many option settings.
- Solution #1 - Dynamic Link Libraries
- Solution #2 - Standard Interfaces
- Solution #3 - System management of component lifetime
- Solution #4 - Registration
- Solution #5 - Interprocess Communication
- Solution #6 - Automation
- Final solution: local and remote plug compatible components.

What's the Problem?

- Building large systems depends on decomposing the logical structure of the system into a hierarchy of components using:
  - class inheritance and aggregation
  - static modular structure
- While establishing an effective hierarchy is essential, it is not enough. The physical packaging of the logical design must:
  - minimize duplication of code
  - minimize compile, link, and load-time dependencies
  - avoid rebuilding large parts of the system when a small change is made to fix a latent error or add new functionality

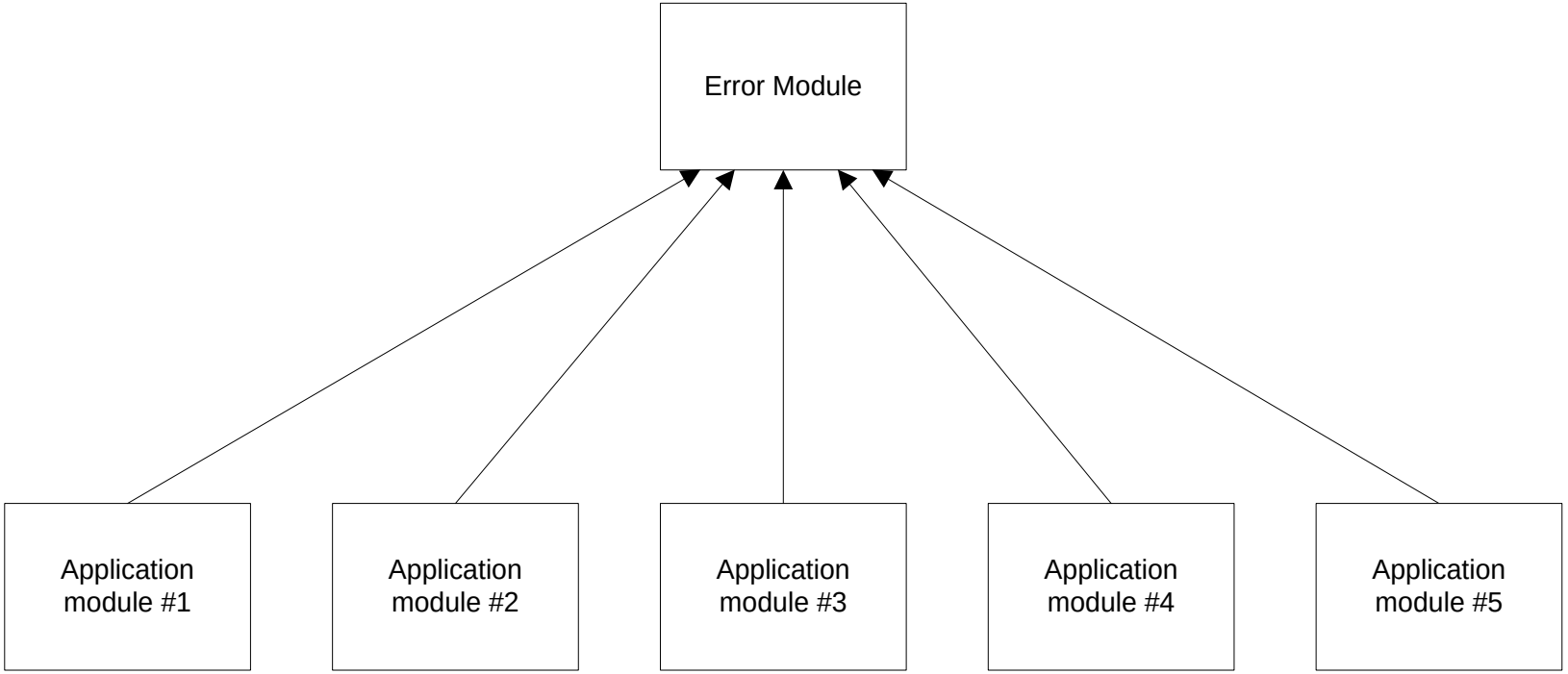
Duplication of Code

- Using conventional technology we build monolithic programs. Each program that reuses a library or module duplicates that code in its execution image.
  - The code occupies disk space for every replicated copy.
  - Two running applications that share source code do not share the corresponding machine code. They each have their own copies that occupy memory in RAM.
  - Since broad reuse of code is an important goal for large systems the duplication of machine code can be a major user of system resources, e.g., memory and load and initialization time.

Compile, Link, and Load-Time Dependencies

- Compile and link time dependencies have been cited as the prime culprit in failures of some very large system implementations, e.g. LargeScale C++ Software Development, John Lakos, Addison-Wesley
  - Dependencies make testing software components in isolation difficult or impossible.
  - Small changes in a single component result in massive recompilation and linking if dependencies are spread out across the system.
  - In large systems parts of a system may be spread over many directories. Then changes to the directory structure cause breakage in compile, link, and load processes.

Compile-Time Dependencies



Rebuilding Components

- Each build requires many files and compile and link options:
  - it is often difficult to ensure that the right versions of source code are included in a build
  - we may not even know all the components required to successfully rebuild a system.
    - Build may take hundreds of files and scores of build scripts and make files.
  - knowing how to set all the options and environments can require detailed knowledge of the design, which for a large system may be very hard to find.
- Sometimes it can be very difficult to find source code of the correct version (supporting the correct platform with all appropriate bug fixes).

What's the Solution?

- Several competing technologies have been invented to package and manage a large system's physical structure:
  - Common Object Request Broker Architecture (**CORBA**) was specified by the Object Management Group (OMG), a consortium of software vendors. Mostly used in UNIX environments for Enter-prise Computing Systems.
  - Component Object Model (**COM**) was developed by Microsoft Corp. and is supported by their development technologies, e.g., Visual Studio with C++, Visual Basic, and Java on windows systems.
  - **JavaBeans**, developed by Sun, Inc. is a modular technology but doesn't fully support physical packaging.
- We will focus on COM because it is available on all current Windows platforms and is being widely used by MS and others.

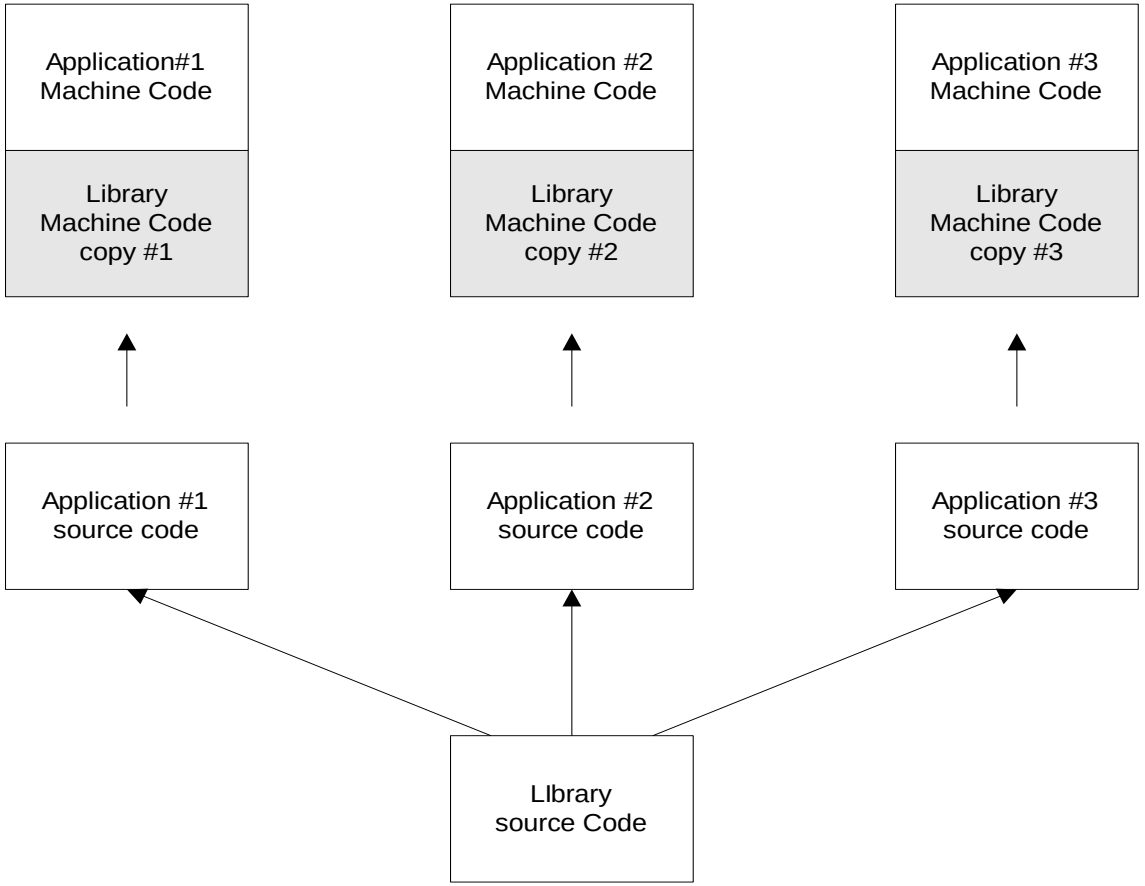
Part #1 of the Solution: Code Reuse by Using DLLs

- Use dynamic link libraries (DLLs).
  - DLLs are loaded at run time from a single file into any running program that needs them, saving disk space for one copy of the object code for each executable that uses the library.
  - DLLs used by several concurrently running executables have only one copy of their code in memory, although each executable maintains local storage for the DLL code. This saves RAM space that would otherwise be required for each running program using the DLL.

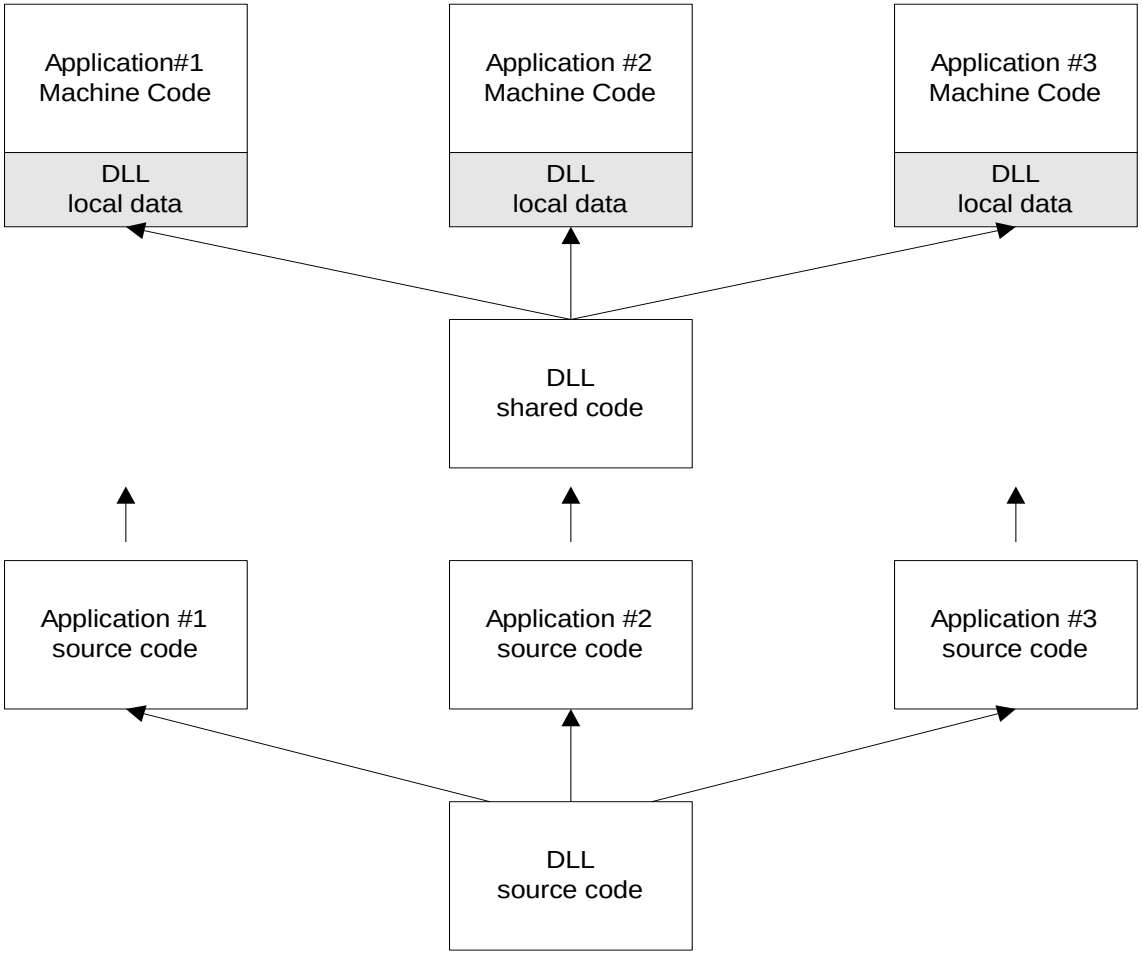
Dynamic Link Library References

- Windows via C/C++, Fifth Edition, Richter and Nasarre, Microsoft Press, 2008
- Windows System Programming, Third Edition, Johnson Hart, Addison-Wesley, 2005
- Win32 Programming, Rector and Newcomer, Addison-Wesley, 1997
- Also, checkout the ProgrammingToInterface Demo in class code directory. It illustrates:
  - How you create and use a dynamic link library
  - How to break compile dependencies

Duplication of Library Code with Static Linking



Sharing of DLL Code



Roadmap

- What's the problem?
  - tight coupling between many components in large systems makes debugging, integration, and maintenance very difficult
  - builds are complex activities depending on many different pieces of source code and many option settings.

√Solution #1 - Dynamic Link Libraries

- Solution #2 - Standard Interfaces
- Solution #3 - System management of component lifetime
- Solution #4 - Registration
- Solution #5 - Interprocess Communication
- Solution #6 - Automation
- Final solution: local and remote plug compatible components.

Part #2 of the Solution: Break Compile-Time Dependencies

- Use component interfaces that carry no implementation detail. You do that by defining interfaces with abstract base classes.
  - Clients see the public member functions but no data members.
  - Components derive from the abstract base class to provide the implementation.

```
struct IInterfaceName {
    virtual void m_fun1(int x)=0;
    virtual char* m_fun2(double y)=0;
}

class implementationName : public IInterfaceName { … }
```

Part #2 of the Solution: Break Link-Time Dependencies

- This comes for free if we use the Part #1 solution. DLLs load at run-time so when a component is recompiled, to fix a latent error perhaps, the client and its other components do not need to be rebuilt, provided there are no compile-time dependencies.
- This helps to make the development process incremental. We can work on each piece, represented by a DLL, in isolation. Then simply run the client to make sure the application works as a whole.
- When a component is revised, we simply copy the new DLL into the directory where the original was stored, overwriting the original. Now when the application is run the new DLL is loaded and we get new functionality without rebuilding other parts of the application.

Roadmap

- What's the problem?
  - tight coupling between many components in large systems makes debugging, integration, and maintenance very difficult
  - builds are complex activities depending on many different pieces of source code and many option settings.

√Solution #1 - Dynamic Link Libraries

√Solution #2 - Standard Interfaces

- Solution #3 - System management of component lifetime
- Solution #4 - Registration
- Solution #5 - Interprocess Communication
- Solution #6 - Automation
- Final solution: local and remote plug compatible components.

Part #3 of the Solution: System Management of Lifetime

- There is one problem with the Part #2 solution.
  - Clients can not instantiate the derived class, which does all the real work of the component, without its header file.
  - But if we give the client the derived class header, we no longer break compile-time dependencies.
- The solution:
  - we could endow the abstract base class with a static creational function which builds the derived object.
  - COM uses an alternate solution: the component supplies a class factory responsible for building any classes derived from the com-ponent's interfaces.
  - The COM library provides a function, CoCreateInstance that clients use to build derived classes using the class factory.
  - Another part of this recipe is to use reference counting to decide when to destroy the component instance.

Roadmap

- What's the problem?
  - tight coupling between many components in large systems makes debugging, integration, and maintenance very difficult
  - builds are complex activities depending on many different pieces of source code and many option settings.

√Solution #1 - Dynamic Link Libraries

√Solution #2 - Standard Interfaces

√Solution #3 - System management of component lifetime

- Solution #4 - Registration
- Solution #5 - Interprocess Communication
- Solution #6 - Automation
- Final solution: local and remote plug compatible components.

Part #4 of the Solution: Registration of Components

- In order to break load-time dependencies, COM provides access to all components through a single point - the windows registry.
- Each component is assigned a Globally Unique IDentifer (GUID) which serves as a key in the windows registry database. Part of the value associated with the GUID key is the directory path to the component.
- Using GUIDs and the registry, clients that need to load a com-ponent do not have to know where it is stored.
  - They simply ask COM to load the component for them by calling CoCreateInstance(…) using the component's GUID.

Roadmap

- What's the problem?
  - tight coupling between many components in large systems makes debugging, integration, and maintenance very difficult
  - builds are complex activities depending on many different pieces of source code and many option settings.

√Solution #1 - Dynamic Link Libraries

√Solution #2 - Standard Interfaces

√Solution #3 - System management of component lifetime

√Solution #4 - Registration

- Solution #5 - Interprocess Communication
- Solution #6 - Automation
- Final solution: local and remote plug compatible components.

Part #5 of the Solution: Interprocess Communication

- Using DLLs work well as long as an instance of a component is used by only one client at a time. However, sometimes it may be important for multiple clients to access the same instance of a component. Perhaps the component is managing information that can be modified by any one of a number of clients, all running at the same time.

- To support this client/server architecture -- one server for multiple clients -- COM provides server “wrappers” for a component that allow it to operate as a stand-alone EXE, communicating with stand-alone clients.

- COM provides a standard method of interprocess communication between client and server called Remote Procedure Calls.

Roadmap

- What's the problem?
  - tight coupling between many components in large systems makes debugging, integration, and maintenance very difficult
  - builds are complex activities depending on many different pieces of source code and many option settings.

√Solution #1 - Dynamic Link Libraries

√Solution #2 - Standard Interfaces

√Solution #3 - System management of component lifetime

√Solution #4 - Registration

√Solution #5 - Interprocess Communication

- Solution #6 - Automation
- Final solution: local and remote plug compatible components.

Part #6 of the Solution: Automation Interfaces

- Automation is a process where scripting languages like visual basic and other languages that do not support C/C++ interfaces can still use COM components.

- It is intended to support, for example, the use of Visual Basic for Applications (VBA) to control COM servers like the Microsoft Office products, e.g., word, excel, access, etc.

- Automation interfaces are provided by the Microsoft Office products and others like Viso. This allows COM designers to use sophisticated processing like viewing complex documents provided by those programs without building the functionality themselves.

Roadmap

- What's the problem?
  - tight coupling between many components in large systems makes debugging, integration, and maintenance very difficult
  - builds are complex activities depending on many different pieces of source code and many option settings.

√Solution #1 - Dynamic Link Libraries

√Solution #2 - Standard Interfaces

√Solution #3 - System management of component lifetime

√Solution #4 - Registration

√Solution #5 - Interprocess Communication

√Solution #6 - Automation

- Final solution: local and remote plug compatible components.

PostScript

- Well, the “Final Solution” is a bit more complicated.
  - Every COM object lives inside a construct called an Apartment.
  - The purpose of the Apartment is to allow a client to use one or more COM objects that may have different threading models or different security models from itself and from each other.
  - The goal is to allow composition of objects without worrying about their implementation details.
  - All communication between apartments occurs via a proxy.
    - Think of the proxy as wrapping a socket or pipe
    - These enable reading and writing bytes, not types.
    - For this reason the COM runtime must know the sizes of each type that travels through the proxy.
    - This is complicated by the fact that different languages may be used to implement the client and the COM objects it uses.

PostScript Continued

- The consequence of using proxies is that COM has to define a binary standard for all types that pass through a COM interface.
- It accomplishes that by adopting the Network Data Representation (NDR) for types.
- This has two consequences:
  - COM interfaces can pass only a limited number of types
  - The interfaces are constructed from an Interface Definition Language (IDL) by an IDL compiler.
- That means that designers need to know how to write IDL and that communication to and from COM objects is not nearly as rich as between objects in any of the conventional OO languages.

End of COM Roadmap