Class Inheritance and OLE Integration (Formerly the Common Object Model)

Technical Overview

Shawn Woods, Mike Vogl, and John Parodi

August 1995

Digital Equipment Corporation
Introduction

This paper defines and compares various forms of inheritance in object systems, and presents a high-level description of the class inheritance mechanism Digital will provide atop OLE Integration (formerly “Common Object Model” or “COM”).

OLE Integration Background

OLE Integration includes the Component Object Model, which is being jointly developed by Digital and Microsoft to provide an open architecture for cross-platform development of client-server applications. OLE Integration will be delivered by Digital in future versions of its ObjectBroker™ product, and by Microsoft in future versions of Windows™ and Windows NT™, making it a strategic technology for object integration from the desktop to servers.

By conforming to the Component Object Model specification, separately developed and maintained application components can operate together correctly. OLE Integration is being defined to provide full compatibility with OLE V2 and beyond, thus these application components may be existing shrink-wrapped OLE applications or custom OLE applications.

In OLE Integration, interoperation between object components occurs through one or more interfaces. An interface is a set of zero or more semantically-related, user-accessible methods. A method is a logical operation provided by an object, and is invoked by sending a message to an object.

Interfaces can inherit from (or extend) existing interfaces. In client-server applications, interfaces provide a clear definition of which services (i.e., object methods) a server provides to its client. The interface is a contract between the client and the object; it defines what the object does but hides the object’s implementation (how the object does it).

A class is a framework for an object's implementation. It defines some parts of the implementation, including what data variables and what functions it uses to perform the methods invoked by its clients. While a class controls some parts of an object's implementation, the class should not be confused with the implementation itself.

In the current specification, OLE Integration’s Component Object Model lacks a strong class mechanism which would allow implementation of new objects by extending existing ones. Digital has announced a program to enhance the Component Object Model by adding a strong class implementation model that prominently features class inheritance.
What is Inheritance?

Inheritance is one of the cornerstones of object oriented programming. Generally speaking, inheritance is the process of using an existing definition and adding to it or changing it. In OLE Integration, there are three types of inheritance: interface inheritance, class inheritance, and implementation or code inheritance.

Interface (Type) Inheritance

An interface is the set of methods supported by a class of objects. (This set is sometimes referred to as an object's type or signature or application programming interface (API).) Interface inheritance is the ability to derive a new interface from an existing one, allowing a new object type to be defined as an extension, or superset, of an existing one. For example, an interface IHuman could be derived from an existing interface IAnimal. The IHuman interface includes all the methods defined for IAnimal because it inherits from it. Inheritance does not deal with the implementation of objects, only with the contract between them and their clients. OLE Integration supports interface inheritance.

Class Inheritance

Class inheritance is the ability to derive a new class from one or more existing classes. It is a way to extend or modify the implementation (code and variables) of an object in a formal way. A class is the formal description of the object’s implementation; class inheritance is a formal process for combining classes.

Generally, class inheritance is used to achieve the following:

- Adding different classes together - deriving a new class from two or more existing classes to build an object that is the combination of the base classes. For example, derived classes CNamedHuman and CNamedDocument could be built from base classes CNamed, CHuman and CDocument.
- Refining a class definition - deriving a new class to modify the base class’s implementation. For example, a derived class CSecureDocument could override a base class CDocument’s implementation of Read and Modify methods to check whether the client has proper authority to take such actions.
- Extending a class - deriving a new class to add new features or capabilities to a base class. For example, the derived class CHuman might add an implementation of the IHuman interface to the base class CAnimal.

Typically, class inheritance provides the following capabilities:

- By default, include all of the implementation definitions/framework of a base class in the derived class
- Allow the derived class to access some portion of the data defined and maintained by the base class
- Provide the client one consistent view of an object, regardless of how many base classes have been inherited
Digital Equipment Corporation
Enterprise Object Software

- Allow the derived class to override the implementation of a base class’s method
- Allow the derived or base class to invoke a specific base class method, even if it has been overridden

In today’s specification, OLE Integration’s Component Object Model supplies neither a formal class mechanism nor class inheritance. Instead, it provides a form of implementation reuse termed aggregation. Aggregation is an informal technique to combine multiple objects into a single pseudo-object at runtime. Aggregation is flexible and eliminates interdependency issues that are difficult to manage in a loosely-coupled environment. However, aggregation has no safeguards against troublesome coding practices, nor does it have programmer tools that support it. While aggregation allows for some code reuse, it gives the programmer no assistance in:

- The coding necessary to create and manage a complex aggregate object
- Refining a portion of an interface
- Extending an interface while reusing the definition of the original interface
- Avoiding invalid or dangerous coding practices when combining and overriding objects in the aggregate

These are complex, repetitive tasks that are typically accomplished automatically by a class inheritance mechanism.

**Implementation (Executable Code) Inheritance**

As mentioned above, a class contains definitions and a framework for an object's implementation, but it does not contain all of the implementation. And at some point, the class must be compiled, turning methods into procedures of executable code. Where this executable code resides depends on the object system in use; conceptually, the location of executable code for an object's methods can be thought of as an "object server." An object server might be a dll, an executable program, or a remote server.

As described above, implementation inheritance means that, by default, the derived class will support the interfaces (i.e., the set of methods) that the base class supports. But how does a derived class obtain access to the executable code that implements the base class's methods?

The answer is that when people speak of an object inheriting implementation or code, they are referring to pointers to code or library entry points. For example, in the case where executable code for methods resides on a server, a derived class inherits a pointer to the executable code that implements the methods. The indirection provided by the pointer means that the derived class may execute the method code, but may not modify it.

These pointers are analogous to global names, which are resolved to memory addresses by the object server, and each method has its own globally-named entry point. This indirection allows the implementation of a method (that is, its executable code) to be
replaced entirely as long as the interface does not change -- because the global names or entry points in the new executable version will remain resolvable by the server. However, if the interface of a base class changes (e.g., by removing methods or by removing or changing the order of parameters) then the derived classes that depend on the old interface will behave unpredictably.

Polymorphism is the characteristic of object systems whereby different implementations of an interface can be passed transparently to a client, which provides the capability to do "generic" operations. For example, a method named display_me could be useful for many objects. However, the code that implements this function would differ greatly depending on the nature of the object to be displayed (e.g., a PostScript file vs. a text file vs. an image). An object server can detect the nature of the object and dispatch control to the appropriate method implementation.

The end result of implementation inheritance and the indirection provided by name resolution in servers is that the client cannot see into or break the object. This gives an object oriented application stability during change and evolution of its component objects.

**Is Inheritance Good or Bad?**

Like most technologies, class inheritance can be used for right or wrong purposes. The determining factors are what inheritance is being used for and what environment it is used within. In general, it can be said that inheritance is a great boon in the software development environment and a potential problem in certain software deployment environments (especially in a distributed system).

Encapsulation is another cornerstone of object technology. An object is a self-contained set of code and variables that will do things for its clients without them having to know how it does it. The object hides, or encapsulates, its implementation from its clients behind one or more APIs (called interfaces in OLE Integration). The object can change its implementation, often radically, without impacting its clients as long as it doesn’t change the interface.

Classes provide another form of encapsulation for objects, in this case between different parts of the object’s implementation. Just as an object is a self-contained unit to its clients, a base class is a self-contained unit to its derived classes.
In client-to-object encapsulation, an OLE Integration interface clearly separates and governs the relationship between the client and the object. But what separates and encapsulates a base class from a derived class? In other words, what insulates a derived class from changes made to the base class, and what protects a base class from having a derived class meddle with and break it?

In standard object oriented languages, the language compiler enforces a set of inheritance rules between classes. Since all the classes being used are written in the same language, there is typically a close association between the derived and base classes; the derived class is allowed to access the base class data variables directly and freely override its methods. The assumptions are that the same developer or development team is writing both the base and derived classes, and that full source code for both classes is available so that problems can be avoided or fixed.

In an OLE Integration environment (which includes the OLE environment) classes are developed in widely varying languages, by completely different organizations and companies. They are deployed in binary format, without source code, and their respective implementations may change radically between versions. Clearly the relationship between derived and base classes in an OLE Integration environment must support a looser association. OLE Integration class inheritance must provide safe ways to combine, refine, and extend base classes without access to their source code, and without depending on any knowledge of their internal implementations.

**Class Inheritance in OLE Integration**

Digital Equipment Corporation has announced and demonstrated technology to add class inheritance to OLE Integration, and has been careful to provide the expected standard inheritance features while also supporting the loose association constraints required in an OLE Integration environment. In general, Digital:

- Provides a rigorous class model applicable to any OLE Integration object implementation
- Uses the OLE Integration Binary Standard between derived and base classes for strong class-to-class encapsulation
- Manages class inheritance and detects/corrects conflicts
Digital Equipment Corporation
Enterprise Object Software

- Supports development of classes in C++ and C (with other languages to be supported over time)

**How OLE Integration Class Inheritance Works**

The OLE Integration class model implements class inheritance using aggregation between base and derived classes. OLE Integration manages the relationship between class contributions to provide the client with a single, consistent view of the entire object. To create an instance of the object, instances of the base classes and the derived class are created and combined. Together these instances build a single, composite object. It is impossible for the client to tell whether the object is from a single class or a composite of base and derived classes, because the object is in a standard binary format.

The figure above illustrates what the implementation for a derived class looks like conceptually. In the figure, the "stick pins" that extend out of the box represent interfaces. The large light-grey box represents the object as seen by the client. The stick pins are the interfaces that a client of the object can use; those contained wholly within the box represent *protected interfaces* between the derived class and the base classes. OLE Integration provides *public* and *protected* interfaces much as C++ provides public, protected, and private access control to methods and variables in a C++ class.

In this example, the derived class CD (the class from which instance iD is created) overrides the interface IB as implemented by the base class CB. This is shown in the figure by the interface IB invoking the instance iD's code. iD has access to the original CB code through a protected IB interface pointer to the instance iB.
OLE Integration's code re-use mechanisms are delegation/containment and aggregation. In delegation/containment, an “outer” object internalizes another, using the “inner” object as a provider of services which the outer object then offers to its clients. The outer object implements the desired parts of the interface and then explicitly passes requests to the inner object. Aggregation is a special case of delegation/containment, wherein the outer object does not have to re-implement the interface (and explicitly pass requests) but simply exposes the interface of the inner object.

Digital's implementation of OLE Integration class inheritance uses aggregation rather than delegation/containment. In other words, a method invocation need not go through the derived class instance and be forwarded to the appropriate base class instance. If an interface is implemented in a base class, and the derived class wants to inherit that functionality, then the interface pointer that is provided by the object utilizes the base class instance directly. There is generally no forwarding of messages from the derived class instance to the base class instance.

Access to a base class instance’s local code and data is via protected interfaces and only when the base class explicitly allows it. The base class must define a protected interface which allows get/set methods on its local data. In this case, the get/set methods need not exactly match the data stored; they only need a logical match. For example, if a base class were to manipulate coordinates, it could provide a protected interface that allowed a derived class to change the coordinate system from cartesian to polar. The get and set methods in the derived class would need to make the translation.

Without class inheritance, problems can arise when one object overrides the interfaces of one or more other objects, e.g., when combining two base classes that support the same interface, when overriding an extended interface of a base class, or when a base class internally links two normally-orthogonal interfaces. Pure aggregation neither brings these problems to the attention of the developer nor provides any mechanism to help solve the problems. The OLE Integration class model structures class inheritance in a way that exposes these issues; it eliminates some and provides optional mechanisms to solve the others.

**Benefits of OLE Integration Class Inheritance**

The OLE Integration Class Model announced by Digital will support all capabilities normally associated with class inheritance:

- The interfaces implemented by base classes, and the base class definitions and framework, are automatically added to the inheriting derived class.
- The derived class can access the base class through public or protected interfaces, giving easy but controlled access to the variables and code of the base class.
- The different interfaces and class contributions are completely managed by class inheritance mechanisms, eliminating the need for the developer to do this complex and repetitive work.
- The derived class can override single methods or complete interfaces of a base class.
The derived and base classes can take a single-object (virtual) view of methods in the object, or select and execute methods on specific classes.

Furthermore, OLE Integration class inheritance has a number of special, industry-leading benefits:

- Base classes may be packaged and deployed in binary form, without source code.
- Derived classes and clients do not have to recompile for new versions or changed implementations of base classes.
- A class can inherit from a base class written in any language - mixed language inheritance is natural and simple.
- Any OLE Integration object can be viewed as a base class and inherited, whether it was written with the OLE Integration Class Model in mind or not.
- The base classes may be remote from the derived class.

**Summary**

The class inheritance mechanisms Digital will provide for OLE Integration, via ObjectBroker products, allow developers to use traditional class inheritance while maintaining OLE Integration’s inherent benefits of binary deployment and freedom from recompiling objects when base classes change implementations. The end result is a rich, comprehensive object system which reduces development and maintenance costs.
Digital believes the information in this publication is accurate as of its publication date; such information is subject to change without notice. Digital is not responsible for any inadvertent errors.

Digital conducts its business in a manner that conserves the environment and protects the safety and health of its employees, customers, and the community.

SOM and DSOM are registered trademarks of IBM Corporation. OpenDoc is a registered trademark of Apple Computer, Inc. Windows and Windows NT are a trademarks of Microsoft Corporation. ObjectBroker, OpenVMS, and the DIGITAL logo are trademarks of Digital Equipment Corporation. CORBA is a registered trademark of OMG.

Comments and suggestions about this paper are welcome; contact John Parodi, 603-881-1640, john.parodi@zko.mts.dec.com). Contact Dan Gilfix (603-881-2317, dan.gilfix@zko.mts.dec.com) or Emile Hebert (603-881-1638, emile.hebert@zko.mts.dec.com) for product and marketing information.

Copyright 1995 Digital Equipment Corporation. All Rights Reserved.