An Object-Oriented Application Framework for DCE-Based Systems

Using the Interface Definition Language compiler and the C++ class library, the HP OODCE product provides objects and abstractions that support the DCE model and facilitate the development of object-oriented distributed applications.

by Mihaela C. Gittler, Michael Z. Luo, and Luis M. Maldonado

HP's Object-Oriented DCE (HP OODCE) provides a library of framework and utility C++ classes that hide DCE programmatic complexity from developers and provide automatic default behavior to ease the development of distributed applications. The default behavior is also a great help in shortening application development time. HP OODCE offers flexibility by allowing developers to use subclassing and customized implementation. Fig. 1 shows the product structure for HP OODCE.

HP OODCE allows clients to view remote objects as C++ objects and to access member functions and receive results without making explicit remote procedure calls (RPCs). Also, applications can communicate with each other using interfaces specified by the Interface Definition Language (IDL). Finally, HP OODCE uses the C++ class library and the IDL compiler (idl++) to create an object-oriented programming environment that supports RPC-based communications, client/server classes, POSIX threads, and access to the DCE naming and security services.

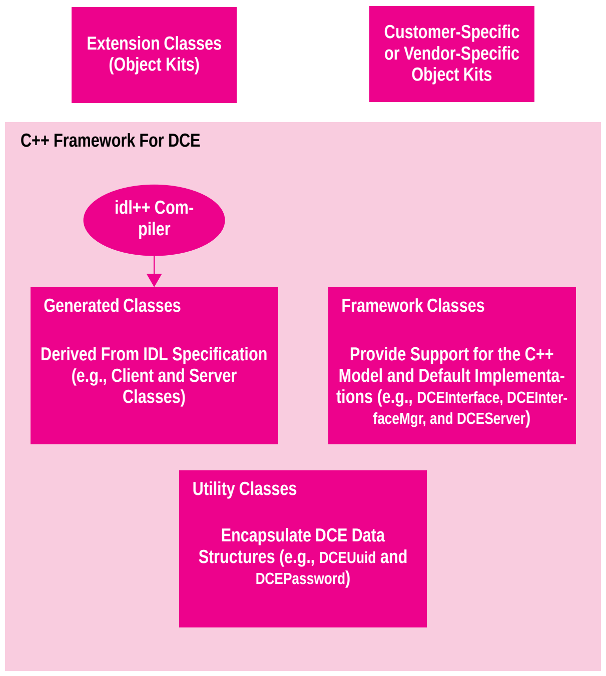

Extension Classes (Object Kits)
Customer-Specific or Vendor-Specific Object Kits
C++ Framework For DCE
idl++ Compiler
Generated Classes
Derived From IDL Specification (e.g., Client and Server Classes)
Framework Classes
Provide Support for the C++ Model and Default Implementations (e.g., DCEInterface, DCEInterfaceMgr, and DCEServer)
Utility Classes
Encapsulate DCE Data Structures (e.g., DCEUuid and DCEPassword)

Fig. 1. HP OODCE product structure.

idl++-Generated Classes

The idl++ compiler takes an IDL specification like the one shown in Fig. 2 and generates the C++ classes shown in Fig. 3. The idl++ compiler also generates the header file and stubs normally produced by the DCE IDL compiler.

The concrete client class* describes the client proxy object that accesses remote C++ objects implemented by the server. The proxy object gives the client the impression that the instantiation of a particular server object is executing locally. Fig. 4 shows an example of a client proxy class declaration for an interface to the Sleep function, which is responsible for putting a process to sleep. This class contains multiple constructors that, when called, locate the compatible manager (server) objects based on location information and the UUID (universal unique identifier) supplied as arguments to the constructors.

The abstract server class in Fig. 3 provides declarations for member functions defined in the IDL specification that correspond to remote operations that can be accessed by the client proxy object. The default concrete server class declares the member functions specified in the abstract class. The functions must be implemented by the application developer. Fig. 5 shows the abstract and concrete server manager declarations for the Sleep function.

The entry point vector contains entry points for each remote procedure defined in the IDL specification.

HP OODCE Server and Client Classes

The server code that interacts with the DCE subsystems is embodied in the DCEServer class. An instance of the DCEServer class, called theServer, manages the remote objects that are exported by the DCE server application. These objects are

```
//foo.idl
[uuid(DOFCDD70-7DCB-11CB-BDDD-08000920E4CC),
 version(1.0)]
interface sleeper
{
[idempotent] void Sleep
   (                          [in] handle_t h
                              [in] long     time),
}
```

Fig. 2. IDL specification for the interface Sleep.

* See glossary on page 60 for a brief description of the C++ terminology used in this article.

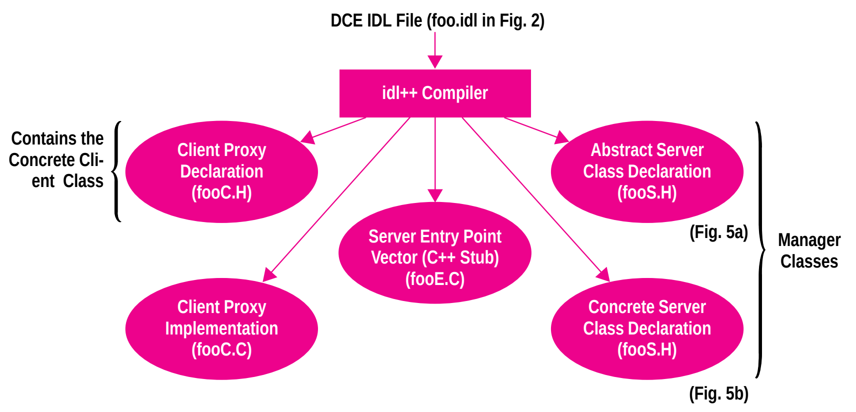

DCE IDL File (foo.idl in Fig. 2)
idl++ Compiler
Contains the Concrete Client Class
Client Proxy Declaration (fooC.H)
Abstract Server Class Declaration (fooS.H)
(Fig. 5a)
Server Entry Point Vector (C++ Stub) (fooE.C)
Manager Classes
Client Proxy Implementation (fooC.C)
Concrete Server Class Declaration (fooS.H)
(Fig. 5b)

Fig. 3. The files created after an IDL specification is processed by the idl++ compiler.

instances of the concrete server manager classes and each has a DCE UUID. There is one DCEServer instance per DCE rpc_server_listen call (currently per UNIX® process), which starts the server's run-time listening for incoming RPC requests. DCEServer has member functions that establish policies such as object registration with the RPC run-time process or the naming service and setting security preferences. Object registration takes place whenever the DCEServer class method RegisterObject is called. Fig. 6 shows the server main program for the Sleep object and the implementation of the Sleep function.

In HP OODCE, server objects are accessed via a client object (see Fig. 7). The client RPC request specifies a binding handle that locates the interface and the DCE object UUID. The entry point vector code locates the correct instance of the requested manager object. Fig. 8 shows the HP OODCE client/server run-time organization.

The idl++-generated client proxy class has methods corresponding to the operations defined in the IDL specification. Idl++ provides an implementation of the client proxy object methods. These methods locate the server and call the corresponding C ++ stub generated by the idl++ compiler. The proxy implementation handles rebinding, sets security preferences, and maps DCE exceptions returned by RPC into C++ exceptions (described below).

```
class sleeper_1_0 : public DCEInterface {
 public:
   sleeper_1_0(DCEUuid& to = NullUuid):
      DCEInterface(sleeper_v1_0_c_ifspec, to) { }
   sleeper_1_0(rpc_binding_handle_t bh, DCEUuid& to = NullUuid) :
      DCEInterface(sleeper_v1_0_c_ifspec, bh, to) { }
   sleeper_1_0(rpc_binding_vector_t* bvec) :
      DCEInterface(sleeper_v1_0_c_ifspec, bvec) { }
   sleeper_1_0(unsigned char* name,
                      unsigned32 syntax = rpc_c_ns_syntax_default,
                      DCEUuid& to = NullUuid) :
      DCEInterface(sleeper_v1_0_c_ifspec, na,e, syntax, to) { }
   sleeper_1_0(unsigned char* netaddr,
                      unsigned char* protseq, DCEUuid& to = NullUuid) :
      DCEInterface(sleeper_v1_0_c_ifspec, netaddr, protseq, to) { }
   sleeper_1_0(DCEObjRefT* ref) :
      DCEInterface(sleeper_v1_0_c_ifspec, ref) { }

   // Member functions for client
   void Sleep(
      /* [in] */ idl_long_int time
      ) ;
};
```

Fig. 4. Client proxy class declaration. The class contains several constructors for the Sleep function. The highlighted constructor is the one used in the examples in this article.

HP OODCE Framework and Utility Classes

The framework classes represent the HP OODCE object model abstraction and provide the basis for DCE functionality and default behavior (see Fig. 9). Classes, such as DCEServer, DCEInterfaceMgr, and DCEInterface interact with DCE through the DCE application programming interface.

The idl++-generated manager classes (server side) inherit from the DCEObj and DCEInterfaceMgr classes. DCEObj associates a C++ object instance, which may export several DCE interfaces, with a specific DCE object. Each DCE object is identified by its object UUID. DCEObj holds the UUID for the DCE object (see Fig. 5b).

```
     class sleeper_1_0_ABS : public virtual DCEObj, DCEInterfaceMgr {
        public:
          // Class constructors must initialize virtual base classes
          sleeper_1_0_ABS(uuid_t* obj, uuid* type) :
①               DCEObj(obj),
                DCEInterfaceMgr(sleeper_v1_0_s_ifspec, (DCEObj&)*this, type,
                                (rpc_mgr_epv_t)(&sleeper_v1_0_mgr)) { }

          sleeper_1_0_ABS(uuid_t* type) :
②               DCEObj((uuid_t*)(0)),
                DCEInterfaceMgr(sleeper_v1_0_s_ifspec, (DCEObj&)*this, type,
                                (rpc_mgr_epv_t)(&sleeper_v1_0_mgr)) { }

          // Pure virtual member functions corresponding to remote procedures
          virtual void Sleep(
                /* [in] */ idl_longint time
                ) = 0 ;
     };
(a)
```

```
     class sleeper_1_0_Mgr : public sleeper_1_0_ABS {
        public:
          // Class constructors pass constructor arguments to base classes
          sleeper_1_0_Mgr(uuid_t* obj) :
③               DCEObj(obj),
                sleeper_1_0_ABS(obj, (uuid_t*)(0)) { }

          sleeper_1_0_Mgr( ) :
④               DCEObj((uuid_t*)(0)),
                sleeper_1_0_ABS(uuid_t*)(0)) { }

          virtual void Sleep(// This is what the developer must implement
                /* [in] */ idl_long_int time
                ) ;
     } ;
(b)

③ Corresponds to ①
④ Corresponds to ②
```

Fig. 5. File fooS.H server-side declarations generated by idl++.(a) An example of an abstract server manager declaration. (b) An example of a concrete server manager declaration.

```
void main( )
{
    try  {           // Handle exceptions from constructor or DCE calls
1       sleeper_1_0_Mgr * sleeper = new sleeper_1_0_Mgr ; // Dynamic UUID
        DCEPthread * exitThd = new DCEPthread(DCEServer : : ServerCleanup, 0) ;

        //  theServer–>SetName( ”/. : /mysleeper”) ;
2       //  Register Sleeper object with server object
        theServer–>RegisterObject(sleeper) ;

        //  Accept all other defaults and activate the server
3       //  Defaults are : Use all protocols, don’t use CDS, no security
        theServer–>Listen( ) ;
    }
    //  Catch any DCE related errors and print out on message if any occur
    catch (DCEErr& exc) {
        traceobj << ”Caught DCE DCEException\ n” << (const char*)exec ;
    }
    //  Destructors are called at this point and take care of DCE cleanup
}
```

(a)

```
//  Developer simply implements one method to provide the implementation

void sleeper_v1_0_Mgr : : Sleep(long int time) {
    // Call the (reentrant!) libc sleep function
    sleep(time) ;
}
```

(b)

1 Instance of Concrete Server Class

2 Register Interface with the Object

3 Setup for Listen

Fig. 6. (a) The server program that handles requests for the Sleep interface. (b) The implementation of the Sleep function.

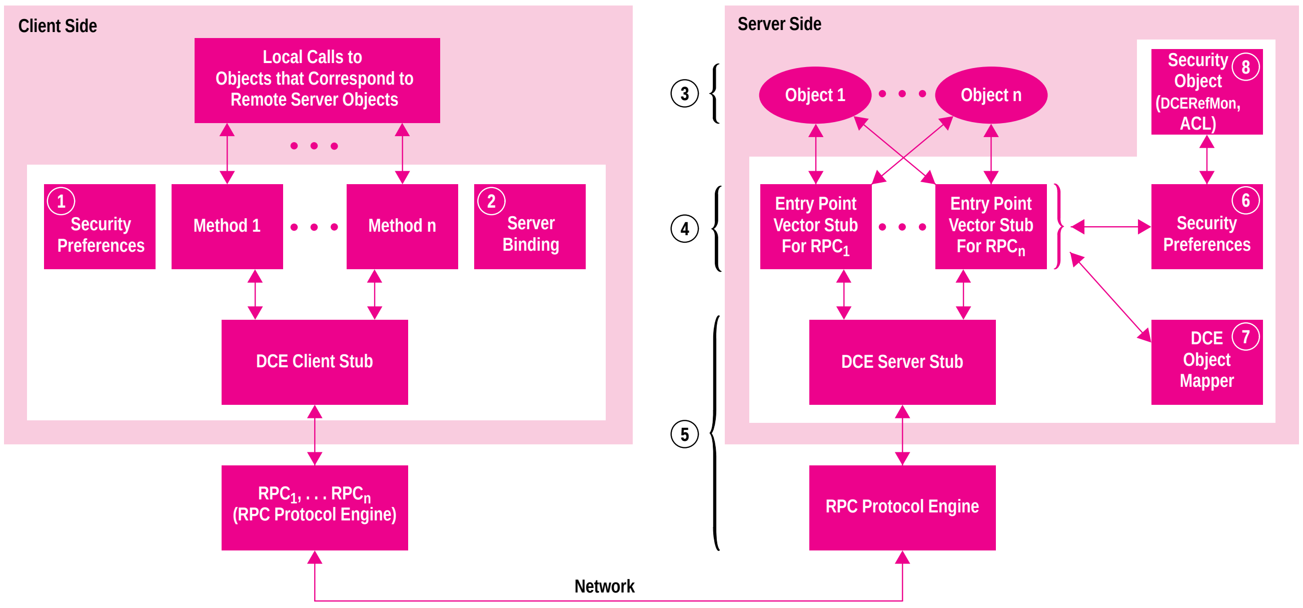
Instance of a Client Proxy Concrete Class

```
main(int, char** argv)
{
    try  {           // Handle exceptions from constructor or DCE calls

        // Constructor takes a network address and protocol sequence
        sleeper_1_0sleepClient( (unsigned  char*)argv [1]
                                (unsigned  char*)“ip”) ;

        // The Sleep method invokes the remote procedure on the server
        sleepClient.Sleep(10) ;
    }

    catch (DCEErr& exec) {
        printf(“DCEException: %s\ n”, (const char*)exec) ;
    }
    exit(0) ;
} ;
```

Fig. 7. A client main program that invokes the Sleep function on the server.

DCEInterfaceMgr is an abstract base class used by the server side of the application to encapsulate object and type information as well as the entry point vector called by the RPC subsystem when an incoming RPC is received (see Fig. 5a). The manager interface is registered with the DCE run-time setup and optionally with the naming service. DCEInterfaceMgr can retrieve the UUID of a particular implementation object instance, the entry point vector, and the pointer to the security reference monitor described by the DCERefMon class.

DCEInterface is an abstract base class used by the client side of the application. This class controls binding and security policies and can retrieve object references. The idl++-generated


Client Side
Local Calls to Objects that Correspond to Remote Server Objects
1 Security Preferences
Method 1
Method n
2 Server Binding
DCE Client Stub
RPC$_1$, . . . RPC$_n$ (RPC Protocol Engine)
Network
Server Side
3
Object 1
Object n
Security Object (DCERefMon, ACL) 8
4
Entry Point Vector Stub For RPC$_1$
Entry Point Vector Stub For RPC$_n$
Security Preferences 6
DCE Object Mapper 7
DCE Server Stub
5
RPC Protocol Engine

1 Sets up security preferences, which have to be compatible with the server’s security preferences

2 Obtains the binding handle to the server

3 C++ object instances defined in the IDL Interface

4 C++ entry point vectors generated by idl++

5 RPC run-time server endpoint and server stub

6 Checks security preferences before allowing the request access to the selected object

7 Locates objects

8 Set up by user

Fig. 8. The HP OODCE client/server architecture.

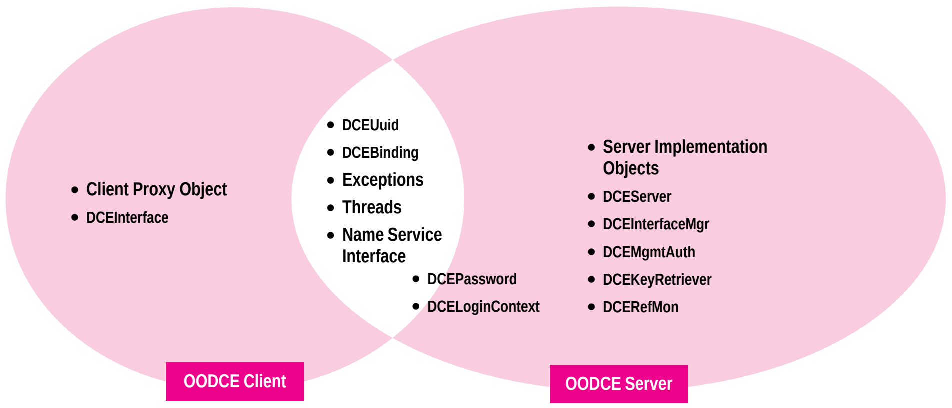


Fig. 9. HP OODCE framework and utility class library components.

client proxy class inherits from the DCEInterface class (see Fig. 4).

The HP OODCE utility classes add convenience to the HP OODCE development environment. These classes encapsulate DCE types and provide direct DCE functionality. For example, DCEUuid deals with the DCE C language representation of the uuid_t type* and its possible conversions to other types, while DCEBinding encapsulates DCE binding handle types.

Other utility classes include:
- Security services: DCERefMon for setting security preferences and DCERegistry for accessing the DCE registry database
- Naming services to model and access objects in the directory namespace
- Thread services to encapsulate the use of pthread mutexes,** condition variables, and thread policies
- Error handling and tracing services to support an exception mechanism and log information.

The security, naming, and thread services are described in the articles on pages, 41, 28, and 6 respectively.

Additional Classes

Additional classes can be derived from the abstract manager class to allow for multiple implementations for a given DCE interface. Each class must be registered with the global server (DCEServer) via the theServer object (remember that theServer is an instance of the DCEServer class). This allows the entry point vector code to locate the object manager instance, verify security preferences, and allow access to the manager methods (see Fig. 8). If the manager object is not immediately located in the HP OODCE internal map managed by theServer object, the entry point vector code can call a user-defined method to activate the manager object according to user-defined polices. Once activated, the manager object is reregistered with theServer and mapped into the object map. An object manager can be deactivated (removed from the object map) when requested by the user application.

While HP OODCE adheres to the object model provided by DCE, two extensions have been made to enhance object functionality. An ObjRef class contains a reference to an object and may be used to pass remote object identities between remote objects. When an ObjRef is used to establish the binding to an object, the referenced object may need to be activated by bringing its persistent data into memory from a file. HP OODCE provides an activation structure that allows this behavior to be implemented easily by the server.

The application developer can add framework or utility classes and provide additional implementations as well as change some HP OODCE default behavior. Additionally, the developer continues to have access to the C language-based DCE API. Direct use of this API is governed only by the correct mapping of exceptions and the corresponding rules for C++ with regard to the C language.

HP OODCE Exception Model

One goal of the HP OODCE system was to create a consistent error model. C++ exception handling was the natural choice as the basis for this model since this mechanism is already well integrated into the language. C++ provides benefits such as object destruction and reduced source code size and is similar in principle to the current DCE exception handling mechanism.

Despite their similarity, the C++ and DCE exception mechanisms do not integrate well. Exceptions raised by one implementation cannot be caught by the other, and more important, those generated by the DCE implementation can cause memory leaks if they are allowed to propagate through C++ code. This latter problem is a result of the use of the setjmp and longjmp functions in the DCE exception implementation, which do not allow run-time C++ to call destructors for temporary and explicitly declared objects before exiting a particular scope.

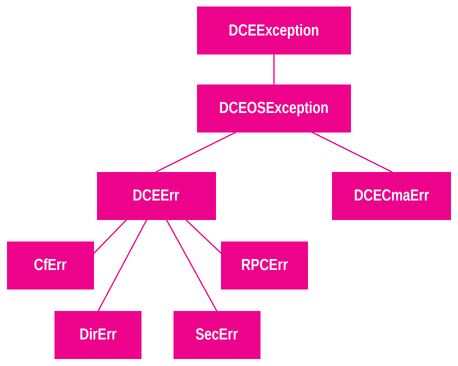


Fig. 10. Exception class hierarchy.

* uuid_t is a C structure containing all the characteristics for a UUID.

** Mutexes, or mutual exclusion locks, are used to protect critical regions of code in DCE threads.

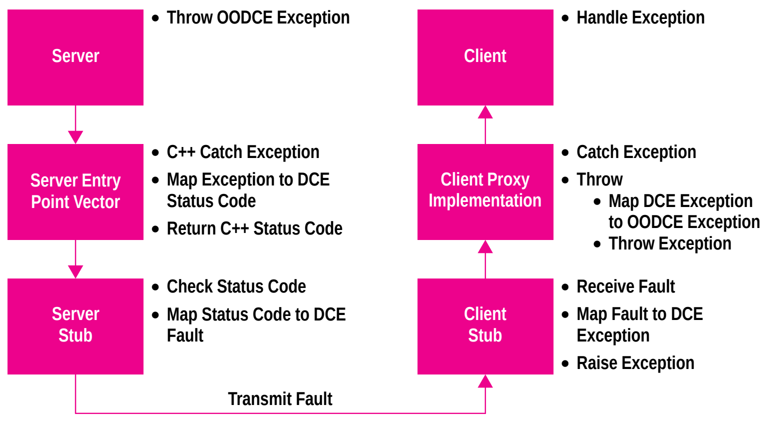


Fig. 11. Exception handling in HP OODCE.

To solve the problems raised by the use of two different exception mechanisms, HP OODCE maps DCE exceptions into C++ exceptions. The HP OODCE classes are arranged into a C++ class hierarchy (see Fig. 10). DCEException is the base class for the hierarchy and provides pure virtual operators to convert exceptions to status codes or ASCII strings. The hierarchy contains subclasses derived from the base class for each of the DCE subcomponents (RPC, security, directory services, configuration, CMA (common multi-threaded architecture) threads, and so on) so that each individual DCE exception can be caught by type.

HP OODCE takes particular care to prevent DCE exceptions from being propagated directly into C++ code. At the boundaries between DCE C and HP OODCE C++ code, DCE exceptions and error status codes are mapped into HP OODCE exceptions and propagated into C++ code. One area that needed particular attention was in passing exceptions between the server and client. We wanted to use the RPC run-time implementation of the server's communication fault transmission, but to do so required a "translation" layer to isolate RPC exceptions from HP OODCE C++ code. This translation layer is implemented within the idl++-generated client proxy implementation and server entry point vector classes (see Fig. 11). C++ exceptions raised in the HP OODCE server are caught in the server entry point vector and mapped to a DCE status code. This status code is then returned to the server stub, which translates the code into a DCE exception and raises it to the attention of the run-time RPC. The run-time RPC takes care of mapping the exception to one of the currently implemented RPC fault codes and then transmits the fault to the client. Basically the reverse happens on the client side, except that here, the client implementation class will catch the DCE exception raised from the client stub and throw the HP OODCE exception back to the client.

Acknowledgments

The idea, overall architecture, initial design, and prototype implementation of this project are mainly attributed to Jeff Morgan. Deborah Caswell, John Dilley, and Bob Fraley each designed and implemented significant parts of the system. We wish to thank them for their continuing advice and contributions to this product. We also wish to thank Amy Arnold, Serena Chang, Jack Danahy, Linda Donovan, John Griffith, and John Stodieck for their effort in making HP OODCE an HP product.

Bibliography

1. J. Dilley, "OODCE: A C++ Framework for the OSF Distributed Computing Environment," *Proceedings of the Winter '95 USENIX Conference*.

2. Open Software Foundation, *OSF DCE Application Environment Specification*, 1992.

3. M.Ellis and B.Stroustrup, *The Annotated C++ Reference Manual*, Addison Wesley, May, 1991.

Glossary

Although the terminology associated with object-oriented programming and C++ has become reasonably standardized, some object-oriented terms may be slightly different depending on the implementation. Therefore, brief definitions of some of the terminology used in this paper are given below. For more information on these terms see the references in the accompanying article.

Abstract Class. Abstract classes represent the interface to more than one implementation of a common, usually complicated concept. Because an abstract class is a base class to more than one derived class, it must contain at least one pure virtual function. Objects of this type can only be created through derivation in which the pure virtual function implementation is filled in by the derived classes.

The following is an example of an abstract base class:

```
class polygon {
public:
     // constructor, destructor and other member functions
     // could go here...
     virtual void rotate (int i)  =  O; //a pure virtual function
     // other functions go here...
};
```

Other classes, such as square, triangle, and trapezoid, can be derived from polygon, and the rotate function can be filled in and defined in any of these derived classes.

Base Class. To reuse the member functions and member data structures of an existing class, C++ provides a technique called class derivation in which a new class can derive the functions and data representation from an old class. The old class is referred to as a base class since it is a foundation (or base) for other classes, and the new class is called a derived class. Equivalent terminology refers to the base class as the superclass and the derived class as the subclass.

Catch Block. One (or more) catch statements follow a try block and provide exception-handling code to be executed when one or more exceptions are thrown. Caught exceptions can be rethrown via another throw statement within the catch block.

Class. A class is a user-defined type that specifies the type and structure of the information needed to create an object (or instance) of the class.

Concrete Data Class. Concrete data classes are the representation of new user-defined data types. These user-defined data types supplement the C++ built-in data types such as integers and characters to provide new atomic building blocks for a C++ program. All the operations (i.e., member functions) essential for the support of a user-defined data type are provided in the concrete class definition. For example, types such as complex, date, and character strings could all be concrete data types which (by definition) could be used as building blocks to create objects in the user's application.

The following code shows portions of a concrete class called date, which is responsible for constructing the basic data structure for the object date.

```
typedef boolean int;
#define TRUE 1
#define FALSE O

class date {
public:
     date (int month, int day, int year); //Constructor
     ~date(l;                            //Destructor
     boolean set date(int month, int day, int year);
     // Additional member functions could go here. . .

private
     int year;
     int numerical_date;
     // Additional data members could go here...
};
```

Constructors. A constructor creates an object, performing initialization on both stack-based and free-storage allocated objects. Constructors can be overloaded, but they cannot be virtual or static. C++ constructors cannot specify a return type, not even void.

Derived Class. A class that is derived from one (or more) base classes.

Destructors. A destructor effectively turns an object back into raw memory. A destructor takes no arguments, and no return type can be specified (not even void). However, destructors can be virtual.

Exception Handling. Exception handling in C++ provides language support for synchronous event handling. The C++ exception handling mechanism is supported by the throw statement, try blocks, and catch blocks.

Member Functions. Member functions are associated with a specific object of a class. That is, they operate on the data members of an object. Member functions are always declared within a class declaration. Member functions are sometimes referred to as methods.

Object. Objects are created from a particular class definition and many objects can be associated with a particular class. The objects associated with a class are sometimes called instances of the class. Each object is an independent object with its own data and state. However, an object has the same data structure (but each object has its own copy of the data) and shares the same member functions as all other objects of the same class and exhibits similar behavior. For example, all the objects of a class that draws circles will draw circles when requested to do so, but because of differences in the data in each object's data structures, the circles may be drawn in different sizes, colors, and locations depending on the state of the data members for that particular object.

Throw Statement. A throw statement is part of the C++ exception handling mechanism. A throw statement transfers control from the point of the program anomaly to an exception handler. The exception handler catches the exception. A throw statement takes place from within a try block, or from a function in the try block.

Try Block. A try block defines a section of code in which an exception may be thrown. A try block is always followed by one or more catch statements. Exceptions may also be thrown by functions called within the try block.

Virtual Functions. A virtual function enables the programmer to declare member functions in a base class that can be redefined by each derived class. Virtual functions provide dynamic (i.e., run-time) binding depending on the type of object.