A PATTERNS-BASED MODEL TO TRANSFER OBJECTS BETWEEN OBJECT-ORIENTED APPLICATIONS

Trần Đan Thư

ABSTRACT

The communication between several applications is an important issue for modern software. Communication protocols are available with many function libraries in support of developing network applications. However, the programming interfaces of communication libraries yield source code fragments to be difficult to reuse because of code duplication or hard coding. Furthermore, database systems also support developers to implement communication functions at a high level. But database systems are only adequate to the development of software systems in which the data storage or administration is most important. In this paper, we propose a light-weight model to give the capability of transferring objects between applications to be implemented in object-oriented programming languages. To realize this model, we apply certain design patterns in designing principal classes to transfer objects from an application to another. A prototype implementation of this project has been developed in C++ as open source.

KEYWORDS

Design Patterns, OOP, Objects Interchange, Communication application.

I. INTRODUCTION

Communication is essential for almost modern software applications. Developers can realize communication functions of software in several ways:

- Programming from scratch, in which developers use traditional APIs and low level protocols to establish connection between applications; for example, the TCP/IP protocol with Socket functions library (Comer and Stevens, 1993);
- Constructing an application by using reusable communication components which were developed from middleware technologies; such as ActiveX, Java Beans, CORBA (Henning and Vinoski 1999; Johnson, 1997);
- Using database systems to establish a high level communication of applications thanks to database records or objects (Elmasri and Navathe, 2006).

The first way is well suited for development of system software packages with important functions at a low level. However, since programming interfaces in APIs (Application Programming Interface) often contain many technical factors as well as implicit semantics, developers who use these APIs may produce source code fragments difficult to reuse and maintain. Such source code fragments are often clumsy, duplicate or hard coded. Moreover, direct use of low level APIs to develop communication functions of an application costs many efforts.

The second way is a promised trend towards a component-based software engineering. Development time and efforts can be reduced because of reusing available components which were proved well working before. Although component-based technologies show an excellent approach of software reuse, there is also a challenge for these technologies. The existence and competition of several component technologies cause considerable obstacles in software development. The problem is that a system developed upon a specific framework may

not work on another. Consequently, a developing software must depend on a specific technological standard, or the heterogeneous problem of technological standards must be resolved.

There are two main options for the third way: relational databases or object-oriented databases. Relational database systems seem not suitable for transferring objects from an application to another because of the gap between relational model and object model: objects are more complex than records. Object-oriented database systems are good selection for storage and transfer of objects. However, because of many existing object-oriented database systems, the selection of a specific system to use is very important (McFarland et al., 2006). Such work is worth only if we are developing a software system in which the data storage or administration is a main function and most important.

On the other hand, GoF design patterns (Gamma et al., 1995) give an advanced object-oriented technique which can be used effectively in software design. Numerous contributions are derived from these patterns to help developers of different application domains, for example, patterns for communication protocols (Byun et al., 2002), patterns for network applications (Schmidt et al., 2002; Sevinç, 2004), patterns for user interface design (Thuy and Thu, 2006). Patterns-based approach is also used to establish infrastructure for frameworks or middleware (Bertrand and Bramley, 2004; Chaumette and Vignégras, 2003; Johnson, 1997; Schmidt et al., 2002). Actually, we are interested in applying design patterns to problems in design of network and communication software (Thu and Tran, 2006a, 2006b).

To support object communication of applications, we propose a light-weight model for interchanging objects between object-oriented applications. Each coming object from sending application must encapsulate both data members and methods such that it has the same behaviour as the original object. Moreover, the model must be independent of a specific technology and easy to realize in an object-oriented programming language. Implementation source code should be succinct and clean, which is reusable as open source to develop object-oriented communication application.

This paper aims to present our current works of the research, in which GoF design patterns (Gamma et al., 1995) are applied to realize the proposed model. The paper is organized in five sections. In section 2, we describe the proposed model for object transfer. Section 3 presents design and implementation of principal classes realizing the model. Section 4 illustrates example of reusing these classes to implement object transfer in simple applications. Finally, section 5 presents our conclusion and future works.

II. PROPOSED MODEL

Figure 1 presents main components of the model for transferring objects of applications. The model comprises three patterns-based layers as follows.

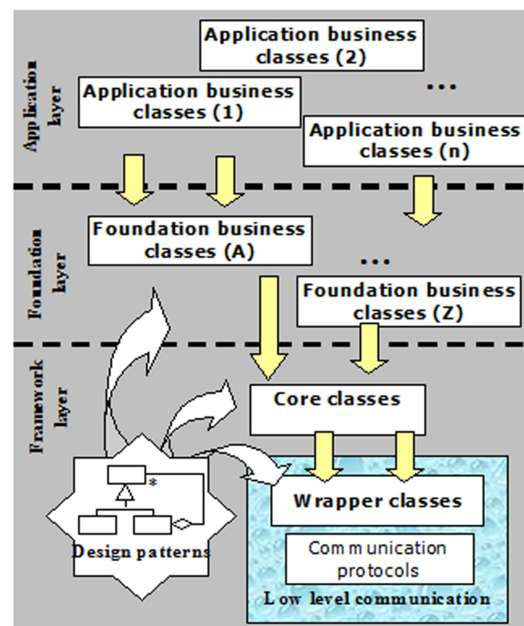

Figure 1: Model for Object Transfer

• Framework layer: This is a fundamental layer establishing basic functions for object communication. The layer is realized by two class systems: wrapper classes and core classes. These classes, which are already implemented and ready for developer's reuse, define programming interfaces at an abstraction level for transferring objects of OO applications;

• Foundation layer: defining classes derived from core classes of the framework layer. The classes of this layer implement requirements of each domain-specific business, which are common and reusable for several applications;

• Application layer: Application developers implement classes for each business application in this layer; they can reuse classes of the two lower layers.

In fact, we need realize in this research only classes of the framework layer. Then we provide examples and guidance to implement classes of the two upper layers. Actually, constructing of reusable class libraries for these two layers costs much effort, which can be realized in an industrial project.

2.1. Low level communication

To establish low level communication, we must resolve firstly raw data transfer between applications. Currently, we use the TCP/IP protocol for sending and receiving raw data because this is a basic protocol which is used widely in almost network applications.

Furthermore, object interchange format is necessary to maintain persistent objects as well as exchange objects with other systems. There are several options for format to interchange objects of applications (Elmasri and Navathe, 2006; McFarland et al., 2006). Recently, XML–based formats are proposed to use for object communication (for example, in Carlo et al., 2005).

2.2. Wrapper classes and Core classes

Wrapper classes aim to encapsulate traditional function libraries for the protocol TCP/IP, and give good programming interfaces to help developer working easily with TCP/IP technique. Developer can create and use communication objects in the developing application, produces reusable and clean source code without entering technical details of the protocol TCP/IP. So we can consider these classes as an abstraction communication protocol.

Core classes are basic classes to implement communication mechanism for instance objects. A concrete class that is derived from a core class has capability of communication, i.e. each instance of this class is an object which can send itself through a communication connection. Core classes are implemented by reusing wrapper classes. Moreover, we also realize a simple system to manage class information. Declaration of a derived class will insert information of the new class into the management system.

2.3. Developer's classes

Developer's classes are constructed at the two upper layers: foundation layer and application layer. Developer derives these classes from classes of the framework layer. A derived class may be general purpose or domain-oriented. Each concrete class derived from a core class can instantiate communication objects.

Communication applications are often organized in two sides: client side and server side. Structure of client application and server application, one of which communicates with each other, is presented in Figure 2. Common business classes which do not depend on client side or server side are reusable for both two sides. The common source code is linked to client application and server application. Because the code is implemented in an object-oriented language, it encapsulates data members and object methods. Consequently, if an object from client application is transferred

to server application, the server side will create a cloning object with the same data members and behaviour as the original object. Actually, in our model, code of methods is available at arrival side rather than to be sent along with transferred object.

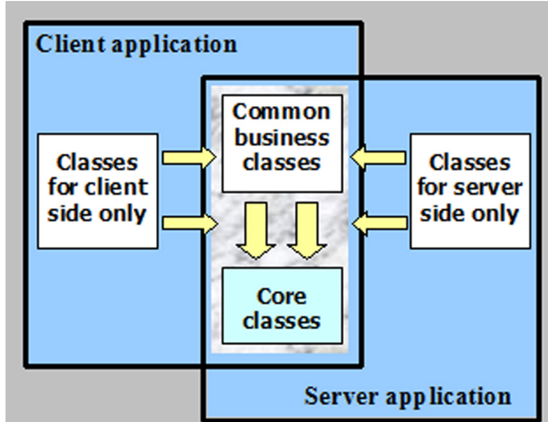

Figure 2: Structure of Client and Server Applications

III. CLASSES REALIZING THE MODEL

This section presents our work to realize the proposed model. We apply design patterns and object-oriented programming technique to resolve directly technical problems rather than using a specific technological standard. This is a solution which does not depend on a specific technology but can give reusable open source.

3.1. Realization of wrapper classes

The wrapper classes are already implemented (Thu and Tran, 2006a). Figure 3 shows UML class diagram describing these classes with relations between them.

• CommPoint class is an abstraction for communication end points, which has two specialized subclasses: ClientPoint class and ServerPoint class. The former is to create a communication object of client application; the latter is to use at server application;

• CommHandle can be considered as a pipe-line for transferring raw data or objects. This is an interface (or an abstract class in C++) which is realized concretely by WinCommHandle class (for Windows) or BsdCommHandle class (for Linux or Unix family);

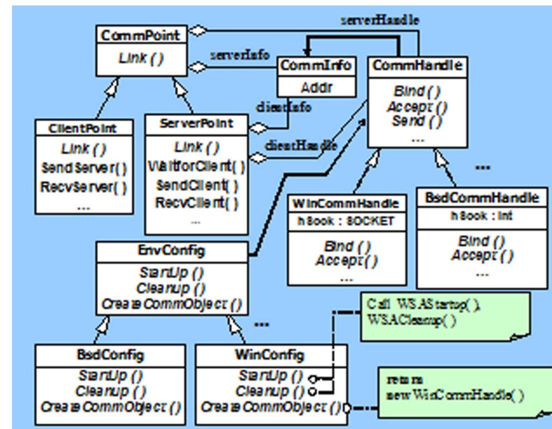

Figure 3: UML class diagram for Wrapper classes (Thu and Tran, 2006a)

• CommInfo class encapsulates certain information for each communication end point such as computer's IP address, or port of communication;

• EnvConfig class (as well as its concrete subclasses) describes a configuration to select dynamically a socket library linked to developing application.

We already implemented and validated wrapper classes by integrating them into network applications. Table 1 presents programming interface of three classes *CommPoint, ClientPoint, and ServerPoint*.

3.2. Realization of core classes

Design of core classes is presented in Figure 4. The main class is NetObject which is an abstract class. This class supports capability of transferring objects through network. NetObject is also responsible for implementing a mechanism of class information management such that any subclass (derived from NetObject) can possess automatically information of this subclass itself.

Classes for basic data types (such as Byte, Integer, Float, String…) implement fundamental operations. Because these classes are derived from NetObject, their instance objects can send themselves through network.

Table 1: Declaration of some wrapper classes

```
class CommPoint {
protected:
      CommInfo* serverInfo; CommHandle* serverHandle;
public:
      int IsValid( );
      virtual int Link( )=0;
      virtual CommHandle* getHandle( )=0;
};  // end of CommPoint declaration
static char* sAddr="255.255.255.255";
class       ClientPoint: public CommPoint {
public:
      ClientPoint(CommHandle* hServer,
            unsigned short port, char*svrAddr=sAddr);
      virtual int Link();
      int SendServer(string buffer, int flag=0);
      int ReceiveServer(string& buff, int flag=0);
      virtual CommHandle* getHandle( );
};  // end of ClientPoint declaration
class       ServerPoint: public CommPoint {
            CommInfo* clientInfo; CommHandle* clientHandle;
      int AcceptClient();
public:
      ServerPoint(CommHandle* hObj, short port);
      virtual int Link();
      int WaitforClients(int backlog=1);
      int SendClient(string buff, int flag=0);
      int ReceiveClient(string buff, int flag=0);
      virtual CommHandle* getHandle();
};  // end of ServerPoint declaration
```

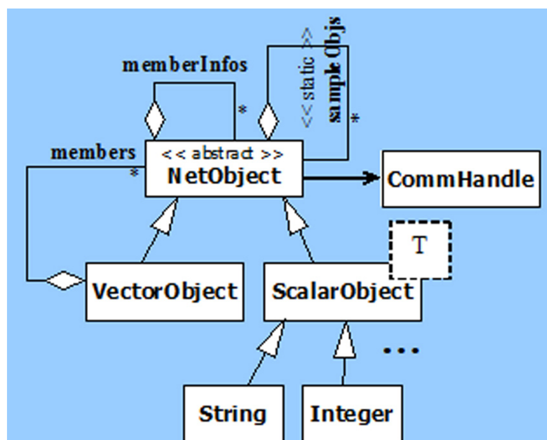

Figure 4: UML class diagram for core classes

Table 2 presents extraction of NetObject declaration. Some members are for internal use only, not for developer's use. Several methods are pure virtual, that are realized in subclasses. However, we already defined two macros: NET_OBJECT_INIT( ) and NET_OBJECT_DECLARE( ), to simplify the realization of these pure virtual methods. Illustration of using these macros is presented in next section.

- The reflection mechanism is implemented for subclasses of NetObject: the static method CreateObj (char*), class level method, is to create an object thanks to the literal class name. For example, the invocation of NetObject::CreateObj("CookieInfo") creates an object of the CookieInfo class.

- The method this->Send(CommHandle* hComm) is to send "this" object through network due to the communication handle "hComm".

- The static method Recv(CommHandle* hComm) is to receive an object. This method can recognize type of the original object to clone a correct object.

- The two methods AddMember() and AddMembers() are used in a derived class

Table 2: Extraction of main core class declaration

```
class       NetObject {
      //------------------------------------   // not for developer
      vector<NetObject*> members;
protected:
      static vector<NetObject*> sampleObjs;
      // some members, not for developer...
      //------------------------------------
      void AddMember(NetObject*) ;
      void AddMembers(NetObject* First, ...) ;
public:
      static NetObject* CreateObj(char* name) ;
      static NetObject* Recv(CommHandle* com) ;
      int Send(CommHandle* hComm) ;
      virtual NetObject* CreateObject()=0 ;
      virtual char* className()=0 ;
      virtual string toString() ;
};  // end of NetObject declaration
```

to save its members into a simple management system of class information. So each object of derived class can manage members of itself.

These above methods are exposed for developer's use. In fact, their implementation is fairly simple, about 100 lines of source code.

3.3. Applying design patterns

We present briefly experience of using GoF design patterns (Gamma et al., 1995) for realizing our model to transfer objects from an application to another.

Following design patterns are used in designing wrapper classes (Thu and Tran, 2006a; Figure 3):

• Adapter pattern is used to adapt different socket libraries to get a common interface CommHandle which does not depend on a specific library. Consequently, source code using CommHandle is reusable independently of native socket libraries.

• Bridge pattern separates the communication concepts (two classes ClientPoint and ServerPoint for client-server communication) from technical details of the TCP/IP protocol which are encapsulated in two classes CommHandle and CommInfo.

• Abstract factory pattern is applied to implement configuration objects (see EnvConfig class and its subclasses) for choosing socket library to be linked to an application running in a specific environment.

• Because there is only one configuration object at run time, Singleton pattern is implemented to ensure that BsdConfig class as well as WinConfig class has unique instance (only one object can be instantiated).

In designing core classes for object communicating, we use three design patterns: Composite, Template Method, and Prototype as follows. These patterns were also used in the initial version of NetObject class which is a case study of distributed design patterns (Thu and Tran, 2006b).

• Composite pattern (see Figure 4) is used to organize hierarchical data structure: the three classes NetObject, VectorObject, and ScalarObject are components participating to an instance of Composite pattern.

• We apply template method pattern several ways in the design to implement methods comprising a process which invokes virtual methods. A template method is able to change suitably its behaviour depending on which object calls the method. For example, the NetObject::Send method

Table 3: A Simple Client Application

```
#include "ClientPoint.h" //ClientPoint class
#include <string>
#include <iostream>
using namespace std ;
int simpleClient(CommHandle* hServer,
            char* sIP, unsigned short port) {
      string data;
      ClientPoint client(hServer, port, sIP);
      client.Link();
      while(1) {
            cout<<"Enter data:";   cin >> data;
            int nByte=client.SendServer(data);
            if(nByte != data.length()+1) {
                  cout << "End connection..." ;
                  return 1;
            }
      }
}
```

```
#include "WinCommHandle.h" //Windows application
void main(int argc, char* argv[]) {
      EnvConfig *cfg=WinConfig::CreateEnvObject( );
      CommHandle* hServer=NULL;
      unsigned short port=4965;
      if(cfg==NULL) return;
      if(cfg->Startup() != 0) {
            cout << "Socket error..." << endl;
            return;
      }
      hServer=cfg->CreateCommObject(SOCK_STREAM);
      simpleClient(hServer, argv[1], port);
      cfg->Cleanup();
}
```

is a template method implementing steps to transfer an object of any type, maybe a scalar, a structured data item, or an array of objects.

• Prototype pattern is used for cloning of objects from sample objects. We use this pattern to recognize type of object which is transferred from sending application.

IV. ILLUSTRATION EXAMPLES

To simplify examples, we use console input and output in programs for purpose of illustration.

First illustration is a simple example of transferring string messages from client application to server application. This example aims to illustrate wrapper classes. Table 3 presents source code of client application.

Client application communicates with its server by using IP address and communication port. In this case, a port (variables or arguments whose names are "port") can take any value from 1000 to 64000 (we choose the number 4965). The main() function is implemented to run on Windows, so a Windows configuration is created by calling the static method WinConfig::CreateEnvObject(). The invocation

cfg->CreateCommObject(SOCK_STREAM) creates a communication handle. The simpleClient() function is to make connection with server application, then sending data:

Table 4: A Simple Server Application

```
#include "ServerPoint.h" //ServerPoint class
#include <string>
#include <iostream>
using namespace std ;
int simpleServer(CommHandle* hServer,
                                     unsigned short port) {
      ServerPoint server(hServer, port);
      server.Link();    server.WaitforClients();
      while(1) {
            string data;
            int nByte= server.ReceiveClient(data);
            if (nByte == 0) {
                  cout << "End connection..." << endl;
                  return 1;
            }
            cout<<"From client:" << data << endl;
      }
}
```

```
#include "WinCommHandle.h" // Windows application
void main() {
      EnvConfig *cfg=WinConfig::CreateEnvObject();
      CommHandle* hServer=NULL;
      unsigned short port=4965;
      if(cfg==NULL) return;
      if(cfg->Startup( ) != 0) {
            cout << "Socket error..." << endl;
            return;
      }
      hServer=cfg->CreateCommObject(SOCK_STREAM);
      simpleServer(hServer, port);
      cfg->Cleanup();
}
```

• creates an object of the ClientPoint class (the "client" variable) ;

• binds this object with server application by invoking client.Link() ;

• sends data by calling client.SendServer().

Similarly, source code for server side is presented Table 4. The server application waits at the specified port (number 4965) to receive data from its client. The simpleServer() function creates an object of the ServerPoint class (the "server" variable), then waits for its client (calling server.WaitforClients() method), receiving data from client (calling server.ReceiveClient() method).

The two functions simpleClient() and simpleServer() are reusable independently of socket libraries. However, the two main() functions (of client and server) are implemented to run with Windows socket library.

Now we present an example of transferring objects. In this example, we implement classes describing two-dimension points, rectangles, and squares. Our purpose is to introduce programming interface for business classes whose instance objects can be transferred through network.

Firstly, we need implement the Point class (see Table 5). In this implementation, two classes NetObject and Integer are core classes which are reused: NetObject is re-

Table 5: Implementation of Point class by reusing Core Classes

```
#include "netObject.h"
class Point:      public NetObject {
      Integer coor_x, coor_y;
public:
      Point(int x=0, int y=0);
      void Move(int dx, int dy);
      NET_OBJECT_DECLARE();
};
NET_OBJECT_INIT(Point);
Point::Point(int x, int y):
            coor_x(x), coor_y(y) {
      AddMembers(&coor_x, &coor_y, NULL);
}
void Point::Move(int dx, int dy) {
      coor_x += dx;
      coor_y += dy ;
}
```

used by inheritance whereas Integer is reused by instantiation.

We can see that the Point class is implemented as normal C++ classes, except the insertion of two predefined macros, NET_OBJECT_DECLARE() and NET_OBJECT_INIT(), to realize automatically certain source code. Furthermore, the instruction AddMembers(&coor_x, &coor_y, NULL) aims to insert two fields coor_x and coor_y of the Point class into the class information management system.

Furthermore, the above implementation permits the Point class send and receive its objects between applications running on network. For example, if p is an object of the Point class and client is a communication point at client side of a client-server connection (i.e. an object of the ClientPoint class), then the instruction p.Send(client.getHandle()) will send p to server application. On the other hand, because we are not yet sure about type of coming object, we use the instruction

NetObject *Obj=NetObject::Recv(client.getHandle( ))

to wait and receive an object; if successful then the Obj variable will hold an object the same as the original one.

Table 6 presents implementation of two classes for rectangles and squares: Rect and Square. We derive the Rect class from an abstract class Shape which is a subclass of the NetObject class, in which the Point class is also reused. The Square class is derived from the Rect class: we need implement only its constructor.

After implementing classes with capability of object transfer, we can reuse these classes in applications which support transferring objects. Table 7 illustrates applications sending and receiving a rectangle. Client application (with the sendObject function) creates an object of the ClientPoint class and stores it in the client variable, then binding this object with server communication handle. An object Rec of the Rect class is created with corner (25, 27) and width=130, height=270. The instruction Rec.Send(client.getHandle()) is to send the Rec object to the server application. Server application (recvObject function) creates object named server of the ServerPoint class. The instruction pObj=NetObject::Recv(server.getHandle()) waits to receive object to be sent from client application; in this case, if successful pObj will hold a rectangle object, an instance of the Rect class, which is a copy of the Rec object.

Table 6: Implementation of some classes derived from Core Classes

```
class Shape: public NetObject {
public:
      virtual Integer Area()=0;
}; // end of Shape
class Rect: public Shape {
      Point corner;
      Integer width, height;
public:
      Rect(Point cor, int ww=0, int hh=0);
      Integer Area();
      void Move(int dx, int dy);
      NET_OBJECT_DECLARE();
};
NET_OBJECT_INIT(Rect);
Rect::Rect(Point cor, int ww, int hh) {
      corner=cor; width=ww; height=hh;
      AddMembers (&corner, &width, &height, NULL);
}
Integer Rect::Area() {
      return width*height ;
}
void Rect::Move(int dx, int dy) {
      corner.Move(dx, dy);
}
```

```
class Square:     public Rect {
public:
      Square(Point cor, int aa=0);
      NET_OBJECT_DECLARE();
};
NET_OBJECT_INIT(Square);
Square::Square(Point cor, int aa):
                              Rect(cor, aa, aa){
}
```

Table 7: Simple Application for transferring objects

```
#include "ClientPoint.h"
void sendObj (CommHandle* hServer, short port) {
      string serverIP ;
      cout <<  "Server IP address:" ;
      cin >> serverIP ;
      ClientPoint client(hServer, port, serverIP);
      client.Link( );
      Point p(25, 27); Rect Rec(p, 130, 270);
      int nByte=Rec.Send(client.getHandle( ));
      cout << "Already sending " << nByte
            << " bytes..." << endl;
}
```

For client application

```
#include "ServerPoint.h"
void recvObject(CommHandle* hServer, short port) {
      ServerPoint server(hServer, port);
      server.Link( );
      server.WaitforClients( );
      NetObject* pObj;
      pObj=NetObject::Recv(server.getHandle( ));
      if(pObj==NULL) return;
      cout<<"My class is "<<pObj->className()<< endl;
      cout<<"Information: "<<pObj->toString()<< endl;
      Shape* pFig=Shape::Cast(pObj);
      if(pFig != NULL) {
            cout<<"My area is "<<pFig->Area()<<endl;
      }
}
```

For server application

V. CONCLUSION AND FUTURE WORK

This paper proposes a simple model for transferring objects of distributed applications. We implemented a prototype in C++ for this model. Several design patterns are used to resolve problems in designing classes realizing the model.

After realizing the prototype, we see that a deep study of design patterns can yield elegant solutions to technological questions for small applications which do not need to use industrial infrastructures. This approach also gives reusable open source which does not depend on specific technologies. Our future work comprises several tracks:

• Use XML–based object interchange formats (Carlo, 2005; Elmasri, 2006; McFarland, 2006) to support object communication with other systems ;

• Develop reusable business classes for the two upper layers in the proposed model ;

• Implement the model in other programming languages which do not support distributed objects ;

• Publish open source packages and documents concerning this model.

REFERENCES

[1] Bertrand, F., Bramley, R., 2004. DCA: A distributed CCA framework based on MPI. In HIPS'04, 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments, IEEE.

[2] Byun, Y., Sanders, B., Chung, K., 2002. A Pattern Language for Communication Protocols. In PLoP 2002, Pattern Languages of Programs conference.

[3] Chaumette, S., Vignéras, P., 2003. A Framework for Seamlessly Making Object Oriented Applications Distributed. In PARCO 2003, Parallel Computing: Software Technology, Algorithms, Architectures and Applications, Dresden, Germany

[4] Carlo, COMBI et al., 2005. Building XML documents and schemas to support object data exchange and communication. In DEXA 2005, International Conference on database and expert systems applications, Copenhagen, Denmark.

[5] Comer, D., E., Stevens, D., L., 1993. Internetworking with TCP/IP: Client-server programming and applications BSD socket version, Prentice Hall.

[6] DONG, T. B. Thuy, TRAN, D. Thu, 2006. User Interface Design by Applying Object – Oriented Design Patterns. In RIVF 2006, Addendum Contributions to the 4th IEEE International Conference on Computer Sciences Research, Innovation & Vision for the Future, February 12-16, Hochiminh City, Vietnam.

[7] Elmasri, R., Navathe, S. B., 2006. Fundamentals of Database Systems, Addison Wesley, 5th Edition.

[8] Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Longman.

[9] Henning, M., Vinoski, S., 1999. Advanced CORBA Programming With C++, Addison-Wesley Longman.

[10] Johnson, R., 1997. Frameworks = Patterns + Components, Communications of the ACM, vol. 40.

[11] McFarland, G., Rudmik, A., Lange, D., 2006. OO Database Management Systems Revisited - An Updated DACS State-of-the-Art Report, Jan. 1999, at www.dacs.dtic.mil/techs/oodbms2/oodbms2.pdf.

[12] Schmidt, Douglas C., Huston, Stephen D., 2002. C++ Network Programming: Mastering Complexity with ACE and Patterns, Addison Wesley Longman.

[13] Sevinç, P. E., Martin-Flatin, J. P., Guerraoui, R., 2004. Patterns in SNMP-Based Network Management. In ICSSEA 2004, 17th International Conference on Software and Systems Engineering and their Applications, Paris, France.

[14] TRAN, D. Thu, HUYNH, T. B. Tran, 2006a. Applying object-oriented design patterns in constructing TCP/IP network applications, Journal of Science and Technology Development - Vietnam National University HCMC, Vol. 9.

[15] TRAN, D. Thu, HUYNH, T. B. Tran, 2006b. Object-oriented design patterns for distributed applications, The Vietnam National Conf. on Selected Issues of Communication and Information technology, Dalat, Vietnam.