OBJECT ORIENTED ANALYSIS AND THE DESIGN OF LARGE CLIENT SERVER APPLICATIONS IN A WINDOWS ENVIRONMENT: AN EXPERIENCE

*K.Z Zamli, **W.A Wan-Hassan, ***N.M Mohd-Zainuddin

**Pusat Pengajian Elektrik Elektronik,*
Universiti Sains Malaysia, Nibong Tebal, 14300 Pulau Pinang

***Consolidated Cable (M) Sdn Bhd, Lot 32, Lebuh Sultan Mohamed 1,*
Bandar Sultan Suleiman,
42000 Port Klang, Selangor

**** Program Pengajian Diploma, Universiti Teknologi Malaysia,*
Jalan Semarak, 54100 Kuala Lumpur
K.Z.Zamli@ncl.ac.uk, zelanx@hotmail.com, norziha@utmkl.utm.my

ABSTRACT

A Unified Modeling Language (UML) is probably the most popular language and notations for Object Oriented Analysis and Design (OOAD) in the industry. In fact, the UML, a unification of James Rumbaugh's Object Modeling Techniques (OMT), Grady Booch's Booch Techniques, and Ivar Jacobson's Object Oriented Software Engineering (OOSE), is fast becoming a lingua franca for software engineers, developers and designers alike. Being a lingua franca, the UML helps software engineers "speak" in the same language. In effect, the UML facilitates reuse of not only codes, but also software architectural designs. In some cases, these architectural designs are also documented as reusable designs or patterns.

This paper, derived from our previous work (Idris *et al.*, 2000; Zamli *et al.*, 1999a; Zamli *et al.*, 1999b; Zamli *et al.*, 1999c; Zamli *et al.*, 1999d; Zamli *et al.*, 1999e), describes our experience using a UML to design large scale object oriented client server database applications in a Windows environment. In doing so, we have developed some reusable designs and conventions in terms of UML class diagrams along with class relationships, cardinality and stereotypes, as well as in terms of component diagrams and their dependencies. Using such designs and conventions, we have incrementally developed a Financial Analysis Module as part of a larger Enterprise Resource Planning Systems using the Borland C++ Builder 4.0, Microsoft SQL Server 7.0 and Rational Rose 98i, in a Windows NT platform with an average of 16,600 lines of codes (LOC) and 98 objects.

While some aspects of the designs and conventions used in the Financial Analysis Module are project specific (e.g. using case diagrams, collaboration diagrams, and sequence diagrams), our experiences indicate that some aspects of the designs can be applicable in other development projects in a similar context (i.e. involving large scale database applications). This paper summarizes some of the lessons learned.

Key words: Unified Modeling Language, Reusable Object Oriented Design

1.0 INTRODUCTION

In the last decade, Object Oriented Analysis and Design (OOAD) techniques for developing software have gained a lot of momentum (Liberty, 1998). OOAD raises some hope of alleviating some of the problems (e.g. design patterns, and code reuse) attributed to ever increasing software complexities.

A Unified Modeling Language (UML) (Rumbaugh et al.,1999) is probably the most popular language and notations for OOAD in the industry. In fact, the UML, a unification of James Rumbaugh's Object Modeling Techniques (OMT), Grady Booch's Booch Techniques, and Ivar Jacobson's Object Oriented Software Engineering (OOSE), is fast becoming a lingua franca for software engineers, developers and designers alike (Liberty, 1998; Muller, 1997). Being a lingua franca, the UML helps software engineers "speak" in the same language. In effect, a UML facilitates reuse of not only codes, but also the software architectural designs. In some cases, these architectural designs are also documented as patterns.

This paper, derived from our previous work (Idris *et al.*, 2000; Zamli *et al.*, 1999a; Zamli *et al.*, 1999b; Zamli *et al.*, 1999c; Zamli *et al.*, 1999d; Zamli *et al.*, 1999e), describes our experience in designing large scale object oriented client server database applications in a Windows environment using UML. In doing so, we have developed some reusable designs and conventions in terms of UML class diagrams along with class relationships, cardinality and stereotypes, as well as in terms of component diagrams and their dependencies. Using such designs and conventions, we have incrementally developed a Financial Analysis Module as part of a larger Enterprise Resource Planning Systems using the Borland C++ Builder 4.0, Microsoft SQL Server 7.0 and Rational Rose 98i, in a Windows NT platform with an average of 16,600 lines of codes (LOC) and 98 objects.

While some aspects of the designs and conventions used in the Financial Analysis Module are project specific (e.g. using case diagrams, collaboration diagrams, and sequence diagrams), our experiences indicate that some aspects of the designs can be applicable in other development projects in a similar context (i.e. involving large scale database applications). This paper summarizes some of the lessons learned.

In the next section, this paper describes the important issues related to our approach. Section 2.0 describes the rationale behind our work. Section 3.0 discusses the UML and reusable design in general. Section 4.0 describes our vision, design constraints and conventions. Section 5.0 outlines our reusable designs. Section 6.0 describes the prototype which uses our proposed designs. Section 7.0 discusses lesson learned and issues arising from our approach. Section 8.0 outlines our future work. Section 9.0 concludes the paper.

2.0 RATIONALE

In everyday life, we are accustomed to patterns. Christopher Alexander, the author of such famous books as "Timeless Way of Building" (Alexander, 1979) and "A Pattern Language" (Alexander *et al.*, 1977) once said:

> *Each pattern describes a problem that occurs over and over again in our environment and then describes the core of the solution to that problem in such a way that you can use this solution a million over times without ever doing the same way twice* (Rising, 1996).

In computer science, people are interested in applying patterns when developing software. Like hardware, software is becoming complex and expensive to develop and manage. Table 1, adapted from Royce (1998), shows a typical expenditure by activity for a conventional large scale software project.

Table 1: Typical Expenditure by Activity

ACTIVITY	COST
Management	5%
Requirement	5%
Design	10%
Coding and Unit Testing	30%
Integration and Test	40%
Deployment	5%
Environment	5%
Total	100%

Developing large scale software can be laborious and notoriously difficult, with the ever present risk of errors creeping in unknowingly. The nature of such developments can also be influential, for example, on the results, costs and controllability that can potentially be achieved.

As can be seen in Table 1, Requirement, Design, Coding and Unit Testing, and Integration and Test activities make up 85% of the total expenditure. Given infinite monetary resources, we will not face any problems. But in the real world, monetary resources are finite. In fact, in most cases, we will want to minimize monetary resources for higher profits and lower costs. We believe that the 85% expenditure on Requirement, Design, Coding and Unit Testing, and Integration and Test activities as shown in Table 1, can be minimized if one can reuse some of the designs in the form of patterns, often termed *design patterns.*

Design patterns can sometimes be viewed as reusable architectures analogous to the architecture of a building. Like the structure of a building, software patterns are abstract and generic solutions to problems that recur perhaps in different contexts.

Because design patterns are abstract and generic, they are not ready made "plug-and-play" solutions. In object oriented terms, software patterns are most often represented in the object by commonly recurring arrangements of classes, and the structural and dynamic connections between them. Patterns are most valuable because they provide a baseline for designers to communicate in. Rather than having to discuss a complex idea from scratch, the designer has to just mention a pattern by name and everyone will know, at least roughly, what is being referred to. In this way, designers can easily communicate their design ideas.

Furthermore, the use of design patterns also reduces the need to reinvent the wheel. The main advantage of design patterns stems from their ability to provide generic solutions to recurrent problems that are peculiar to particular situations (Bosch, 1998).

Nevertheless, there is no silver bullet in software engineering (Brooks Jr., 1987; Voas, 1999). Any design patterns should, therefore, be evaluated (i.e. successfully applied to design real world software) before they can be considered as patterns. This can be seen in the work by the so-called Gang of Four (GoF), consisting of Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides (Gamma et al., 1994). They have provided some ground work in identifying common design patterns for software systems.

In terms of our current work, it may be a little too early and too ambitious to consider our reusable designs and conventions (i.e. in terms of UML class diagrams along with class relationships, cardinality and stereotypes, as well as in terms of component diagrams and their dependencies) as design patterns. This is because further experimentations are needed to use the designs in many other different contexts. Nonetheless, we believe that this is a step in the right direction.

3.0 UML AND REUSABLE SOFTWARE DESIGNS

There are a number of ways in which reusable designs may be described (such as reusable designs communicated in a natural language narrative (Amnon et al., 1997)). In terms of our work, it seems natural to adopt the UML because of its popularity. Furthermore, it is a widely accepted modeling language that supports object oriented analysis and the design of software systems.

The UML is a language and notation for specification, construction, visualization and documentation of models of software systems (Rumbaugh *et al.*, 1999). It is fast becoming an industrial standard (i.e. version 1.3) since its adoption by the Object Management Group (OMG) in 1994 (Liberty, 1998; Muller, 1997). Historically, the UML is the result of work done by the following three amigos; James Rumbaugh, Grady Booch and Ivar Jacobson.

The UML provides nine types of diagrams to represent object oriented software designs. These diagrams consist of Use Case Diagram, Sequence Diagram, Class Diagram, Collaboration Diagram, Object Diagram, State Chart Diagram, Component Diagram, Activity Diagram and Deployment Diagram. Each of these diagrams makes up the so-called 4+1 View.

The 4+1 Views consists of Use Case View, Logical View, Implementation View, Process View, and Deployment View provides a cohesive binding among different phases of the software lifecycles (i.e. specification, analysis and design).

The Use Case View specifies the behavior and surroundings of the system in terms of use cases and actors. The Logical View, on the other hand, describes the logical structure of the system, that is, the classes and their relationships. The Implementation and Process View describe the physical structure of the system in terms of how the system is divided into executable files (e.g. exe files) and how the dynamic link libraries (DLL) are structured (e.g. source codes). The Deployment View shows the interconnections between the system's processors and devices, and the allocation of process to processors in its designated environments.

Rarely does one see similarities (or patterns) in the Use Case View. This is due to the fact that requirements and specifications tend to be different for different projects. Unlike Use Case View, Logical, Implementation and Process Views, can have reusable designs (e.g. as design patterns), in particular, class diagrams and component diagrams (Idris et al., 2000). Nonetheless, one must be aware of the contexts that these designs were constructed and their constraints.

4.0 VISION, DESIGN CONSTRAINTS AND CONVENTIONS

Our long term objective is to develop a client-server database application for a Windows environment that employs conventional graphical user interfaces (e.g. windows, icons, menus and pointers) developed from reusable designs and conventions. The application must allow both human to computer interactions and computer to database interactions. The interaction between computers to database is done through the Open Database Connectivity (ODBC).

To facilitate design readability, it seems reasonable to adopt a predefined set of design conventions. Design conventions include specifying variable names, source file names, class names and their relationships. For our class designs, we adhere to the following convention. We stereotype a class that implements the human to computer interaction, a boundary class. A control class is a class that implements computer to database interaction. A utility class is a class which implements useful miscellaneous subroutines. An entity class is a class which abstracts the actual database table. It should be noted, however, that these stereotypes do not carry any special meaning. In addition, we have also used the common Hungarian notation for our class names (i.e. our class names will start with a capital T).

There are two common ways which reusable designs or patterns can be conveyed, namely the Alexandrian form and the GoF form (Appleton, 1997). The Alexandrian form is based on the form proposed by Christopher Alexander (Alexander et al., 1977) to describe his pattern. Alexandrian form dictates that every pattern must provide the following items (Coplien, 1998):

- The pattern name
- The problem the pattern is trying to solve
- Context
- Forces or tradeoff
- Solution / Examples
- Resulting context
- Design rationale

The GoF form is based on the work of Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides (Gamma et al., 1994). GoF form patterns dictate that every pattern must provide the following items:

- Pattern name and classification
- Intent
- Other well known name
- Motivation
- Applicability
- Structure
- Participants
- Collaboration
- Consequences
- Implementation
- Sample Code
- Known Uses
- Related Pattern

The detailed discussions of each item in both representations (Alexandrian versus GoF forms) and their pros and cons is beyond the scope of this paper. In this paper, we are opting for the Alexandrian form because of reasons described in Idris et al. (2000). Now that some conventions are in place, our reusable design patterns can be conveniently described.

5.0 THE PROPOSED REUSABLE DESIGN

Unlike DOS applications, Window applications tend to have longer source codes (mainly to accommodate graphical user interfaces). It seems to be a common practice for Window developers to use the dynamic link library (DLL) as a way to manage source codes. The main advantage of a DLL is that it can be loaded dynamically at run time and can be made independent from any particular applications (i.e. DLLs can be shared amongst many applications). DLL also simplifies the chores of bug fixes and library updates (i.e. without the need of reinstalling the application).

The first reusable design is the main program. Table 2 describes the design in details.

Table 2: Main Program

Pattern Name	**Main Program**
The problem the pattern is trying to solve	Organizing dynamic link library in a structured manner to assist traceability of requirements against source codes
Context	No preconditions, applicable to any situations
Forces	None
Solution and Examples	See Figure 1 and Figure 2
Resulting Context	Applicable only for designing database applications in the windows environment because it uses DLLs.
Design Rationale	See discussions in the next paragraph

The main program is made up a class TMainPanel and one or more TDynamicLinkLibrary classes. TMainPanel class has a boundary stereotype whilst TDynamicLinkLibrary has a utility stereotype as depicted in Figure 1 below.

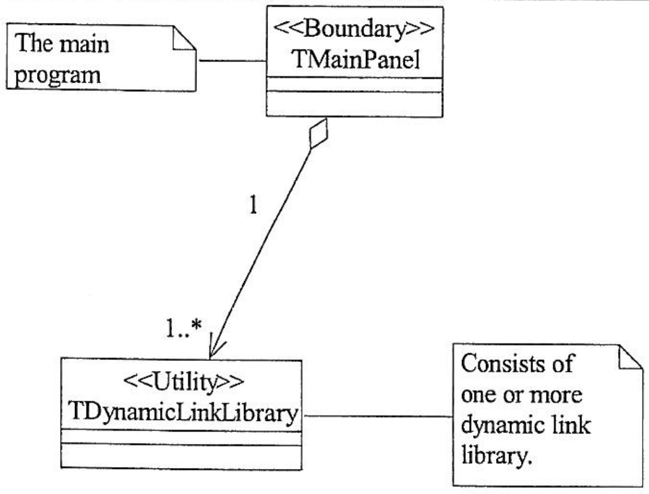
The main program
<<Boundary>> TMainPanel
1
1..*
<<Utility>> TDynamicLinkLibrary
Consists of one or more dynamic link library.

Fig. 1: Class Diagram of TMainPanel and its dynamic link library components

The main panel or the main program interacts with one or more DLLs. DLL instances are implemented as protected fields with restricted aggregation to the main panel. Such relationship provides a loose coupling between the main and its corresponding DLLs (Idris et al., 2000). In other words, the lifetime of each DLL and its main are independent of each other. In effect, this permits the same DLL to be used by more than one application.

Furthermore, such relationships also encourage polymorphous behavior of the object via inheritance at a later stage of the development or in the future upgrades. Figure 2 depicts such a possible polymorphous update.

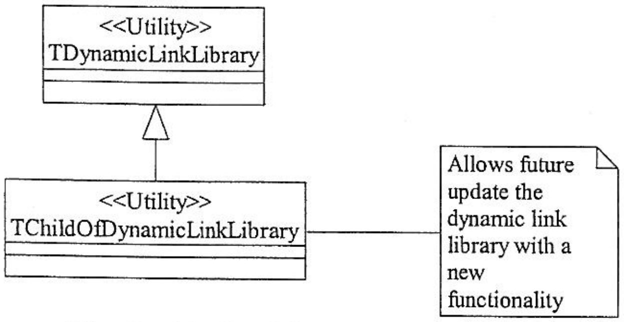
<<Utility>> TDynamicLinkLibrary
<<Utility>> TChildOfDynamicLinkLibrary
Allows future update the dynamic link library with a new functionality

Fig. 2: Derived dynamic link library

Despite its benefit, developing a DLL can be a daunting task. To ease such difficulties, we have defined a design pattern for a DLL. Table 3 summarizes the design pattern in details.

Table 3: Dynamic link library design

Pattern Name	**Dynamic Link Library Design**
The problem the pattern is trying to solve	Separation of concern amongst classes performing different functionalities.
Context	No preconditions, applicable to any situations
Forces	None
Solution and Examples	See Figure 3
Resulting Context	Applicable only for designing database applications in the Windows environment because it uses DLLs.
Design Rationale	See discussions below.

The design for our dynamic link library can be seen in Figure 3. It is only visible to the TDynamicLinkLibrary. In turn, TDatabaseManager and TListClass are visible to the TInterfacePanel. The TDatabaseManager is composed of one or more TDatabaseTables.

In summary, our design for dynamic link library has the following characteristics:

- Each class is loosely coupled and implements only specific functionality (e.g. Database access is only done by an instance of TDatabaseManager).
- An instance of database manager class (TDatabaseManager) can be implemented privately as part of the interface panel (i.e. polymorphous behavior is not desirable).
- DLL is a standalone class that exports at least one major group of functionality in a particular Use Case.
- All DLLs instances are considered to be part of the main panel or main program with restricted navigation.
- All the entity classes are part of their respective control class managers.

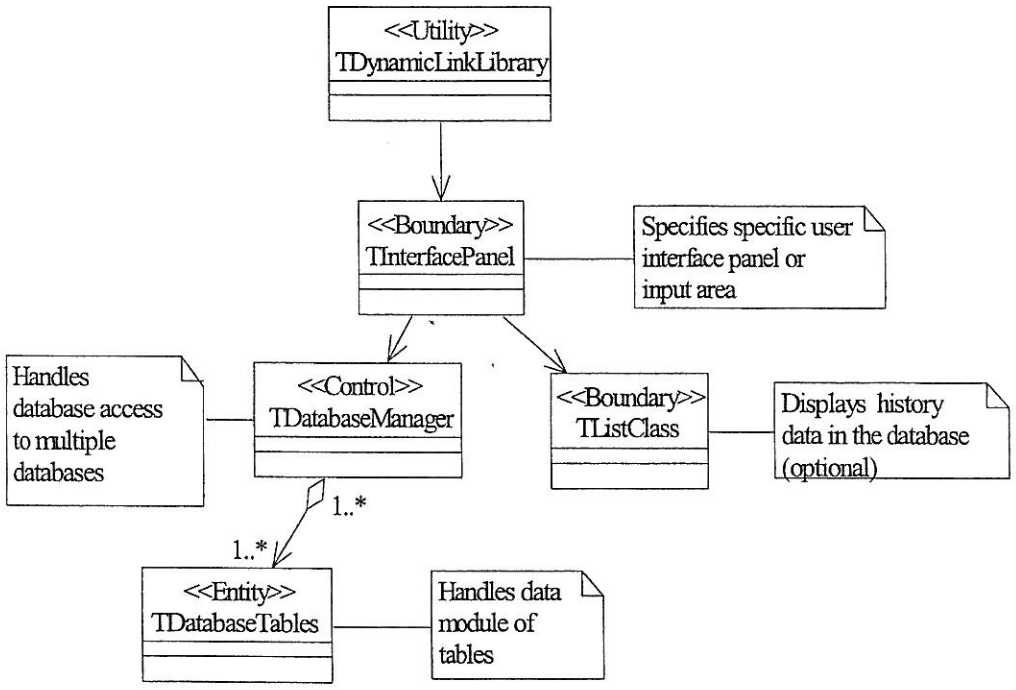
<<Utility>>
TDynamicLinkLibrary
<<Boundary>>
TInterfacePanel
Specifies specific user interface panel or input area
Handles database access to multiple databases
<<Control>>
TDatabaseManager
<<Boundary>>
TListClass
Displays history data in the database (optional)
1..*
1..*
<<Entity>>
TDatabaseTables
Handles data module of tables

Fig. 3: Dynamic link library classes and relationships

Table 4: Source Code Dependencies

Pattern Name	**Code Dependencies**
The problem the pattern is trying to solve	Managing large source codes into well defined modules based on dynamic link libraries
Context	Depend on implementation language. To adopt this pattern, C++ must be used.
Forces	Related to pattern given in Figure 1. Must be used together with pattern described in Figure 1.
Solution and Examples	See Figure 1 and Figure 4
Resulting Context	Applicable only for designing database applications in the Windows environment because it uses DLLs.
Design Rationale	Improves code readability

Figure 4 depicts dependencies amongst various components in C++. The package body and specification for DLLs must be replicated for each of the DLLs depending on the multiplicity defined in Figure 1.

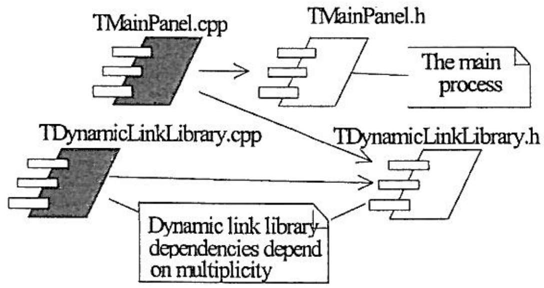

Fig. 4. Component diagram dependencies

An aspect which does not form part of our reusable designs is class operations (i.e. methods) and their attributes. Our experience indicates that operation and attributes tend to be different amongst classes (except the common set and get operations) even the ones sharing the same stereotypes. As a result, operations and their attributes are left out (i.e. to be determined during the actual implementation).

6.0 PROTOTYPE

Using the reusable designs and conventions described in preceding sections, we have developed a prototype Window based client server database application, called the Financial Analysis Module, in a Windows NT environment. The development was done as part of our 9-month long MSc industrial attachment at Consolidated Cable Malaysia, Sendirian Berhad and the Centre For Advanced Software Engineering, Universiti Teknologi Malaysia. Essentially, the Financial Analysis Module deals with day to day accounting and management in an industrial setting.

We have chosen Borland C++ 4.0 for our development partly because of our customer requirements. In addition, Borland C++ 4.0 also offers Rapid Application Development (RAD) capabilities. As far as the database is concerned, our prototype Financial Analysis Module uses Microsoft SQL 7.0. It should be noted that while we have used Microsoft SQL 7.0, our database design is actually open. Any relational database can be used as long as the chosen relational database supports the Open Database Connectivity (ODBC) alias through TCP/IP. In terms of supporting the UML tool, we have opted for Rational Rose 98i because it supports round trip engineering from inception to transition phase; we also use the iterative and incremental software development lifecycle.

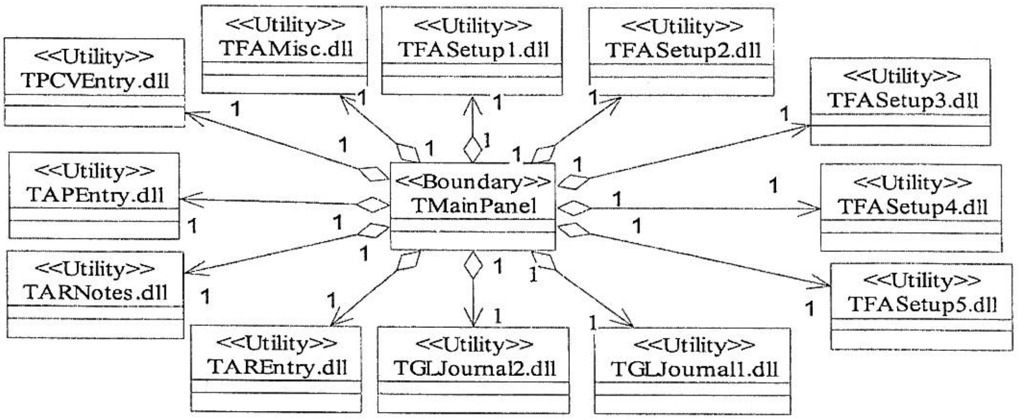

Fig. 5: Financial Analysis Module and its dynamic link libraries

Figure 5 depicts our main program utilizing our reusable designs and conventions, called Main Program in Table 2. In this case, we have decided that the Financial Analysis Module is to be composed of 12 main dynamic link libraries. As illustration, figure 5 depicts our prototype Financial Analysis Module and its dynamic link libraries.

As a rule of thumb, each of the dynamic link libraries is only allowed to elaborate and implement at most two use cases, which are related to each other. We believe that such efforts should encourage improved source code management.

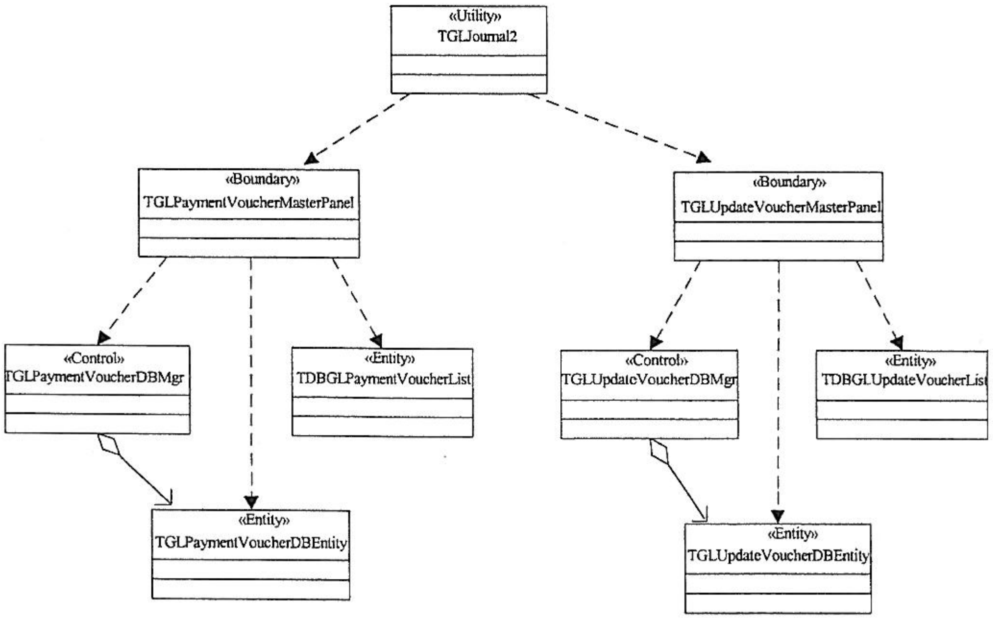

Figure 6: TGL Journal2 dynamic link library and its dependency classes

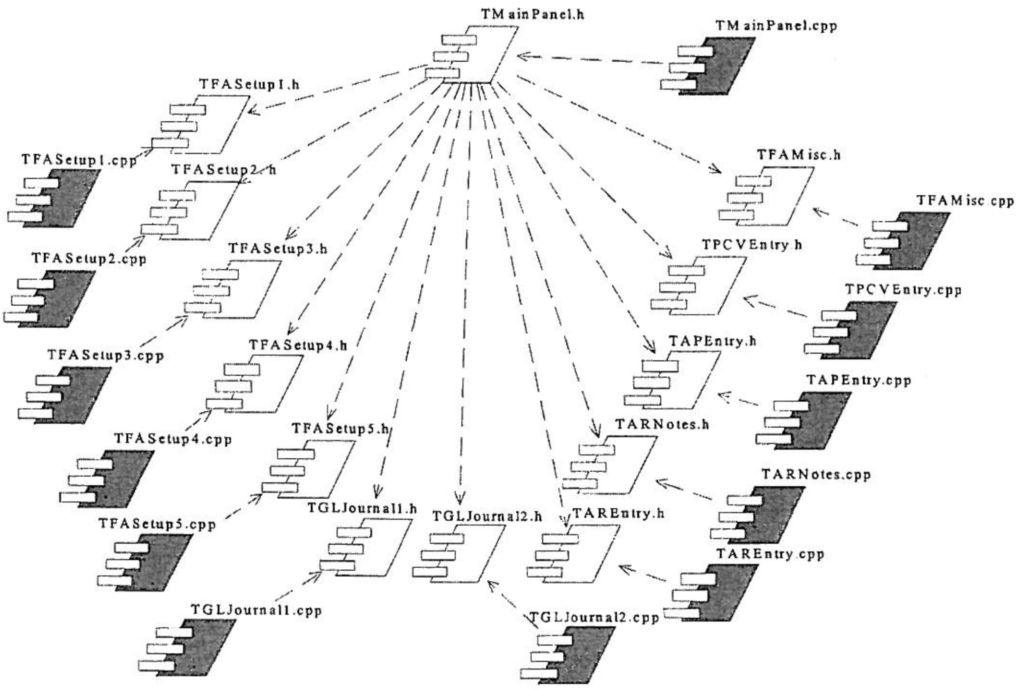

Figure 7: Component dependencies in terms of source codes

Statistically, the Financial Analysis module implements 23 Use Cases (with 98 collaborating objects, 16607 average lines of codes (LOC) and 40 database tables as shown in Table 5.

Table 5: Various implementation statistics

Dynamic Link Libraries	Use Cases Handled	Collaborating Objects	Average Lines of Codes (LOC)	No of DB Tables
TmainPanel	1	1	620	40
TFASetup1.dll	2	10	1375	
TFASetup2.dll	2	10	1265	
TFASetup3.dll	2	10	1370	
TFASetup4.dll	2	10	1272	
TFASetup5.dll	1	5	605	
TGLJournal1.dll	1	5	990	
TGLJournal2.dll	2	10	1870	
TAREntry.dll	2	10	1570	
TAPEntry.dll	2	10	1600	
TARNotes.dll	2	10	1790	
TPCVEntry.dll	2	10	1830	
TFAMisc.dll	2	2	450	
TOTAL	**23**	**98**	**16607**	**40**

To demonstrate our implementation of a dynamic link library (using the reusable design called Dynamic Link Library Design in Table 3.0), Figure 6 in the previous page depicts sample classes involved in the dynamic link library called TGLJournal2.dll. It is worth noting that all other dynamic link libraries are implemented in the same way.

In terms of code generation, Figure 7 in the previous page represents our overall program structure in terms of component dependencies. This is derived from component diagram dependencies (called Code Dependencies in Table 4.0).

Figure 8 depicts sample snapshots of our Financial Analysis Module. As for application functionality, our prototype Financial Analysis Module has the following features:

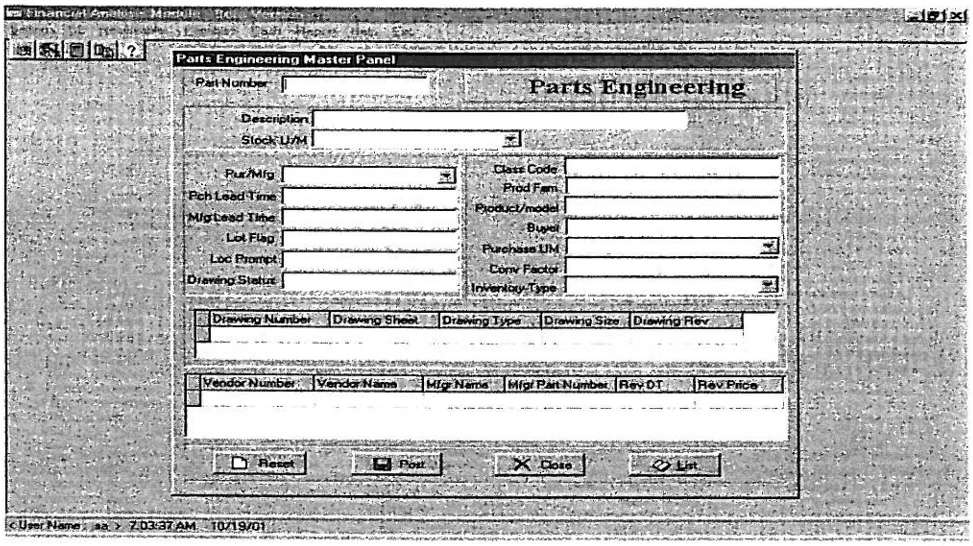

Figure. 8 Sample Snapshots of our Financial Analysis Module

- Support multiple Charts of Accounts
- Support generation of Trial Balance
- Support unlimited no of closing of accounts per financial year
- Support transaction in multiple currencies.
- Support database cache features in the client terminal.
- Support database cache features in client terminal.
- Support multiple parts accounting and part numbering.
- Support multiple access level, which is centrally controlled.
- Support customizable help display.
- Support local as well as Client-Server database access.
- Support tracking of partial repayment,
- Support retrievable history of transactions

7.0 DISCUSSION AND LESSON LEARNED

Firstly, the main lesson learned from this work is that not all UML diagrams are reusable, in particular, use case diagrams, sequence diagrams and collaboration diagrams tend to be requirement specific. This is expected as requirements for different projects tend to be different.

Secondly, the Financial Analysis Module suggests that our reusable designs and conventions may be used in other large client server Windows applications although with different methods and message flows. This is because our designs and conventions are developed with reusability in mind. Thus, we delegate many of the software functionalities to different modules mainly using DLLs. The modules, in turn, can also be expanded via polymorphism. This is demonstrated by our implementation of DLLs for the Financial Analysis Module.

In fact, any applications built in this way can have as many DLLs as possible, provided that garbage collection is handled properly. DLLs must be unloaded once it is not used so that no memory leak can occur. The flexibility of DLLs allows application to be developed by many developers independently and there is almost no limit with respect to software lines of codes (LOC).

Thirdly, one issue which may be raised is the level of granularity of our designs. While it may be useful to present the designs at the very fine grained level of granularity, a counter argument suggests that such level of granularity may affect generality of the designs. As a result, such designs may only be applicable only to certain similar software development projects.

Lastly, the main limitation of our work is the fact that we extensively assume that the operating systems would support the modularization and sharing of codes using dynamic link libraries. This can be somewhat limiting because the concepts of dynamic link libraries are only applicable in the Windows environment. However, this is not to say that the reusable designs and conventions described here are unusable in other operating systems. This reason is that in other operating systems, we can always make some intelligent work, for instance, by converting these dynamic link libraries into accumulated object codes which can then be accessible as conventional software libraries.

8.0 FUTURE WORK

The Financial Analysis Module presents our first successful attempt to use reusable designs and conventions; it is perhaps too ambitious to consider them as design patterns.

Currently, our prototype is undergoing on line testing with real data at our industrial partner's site. The general feedback from the users has been encouraging.

Our ongoing work is to adopt our designs and conventions to assist other in-house development projects for our industrial partner; in particular, we would like to adopt the design patterns for developing Human Resource Management Modules and Enterprise Resource Planning (ERP) Modules. The second co-author of this paper is currently involved in the project.

9.0 CONCLUSION

In conclusion, we have proposed reusable object oriented designs and conventions for developing large client server database application in a Windows environment. This is done in terms of the logical, implementation, and process views. The designs actually consists of UML class diagrams along with class relationships, cardinality and stereotypes, as well as component diagrams and their dependencies.

While some aspects of the designs and conventions used for example, use case diagrams, collaboration diagrams, and sequence diagrams, our experiences indicate that some aspects of the designs can be applicable in other development projects in a similar context (i.e. involving large scale database applications), even ones which require more lines of codes.

ACKNOWLEDGEMENT

We would like to thank all staff of the Centre For Advanced Software Engineering, Universiti Teknologi Malaysia for their help particularly Professor Dr Norbik Idris and all staff of Consolidated Cable Malaysia Sendirian Berhad particularly En Idris Puteh, En Abdul Rahim Jaffri, and En Mohd Yusmin Mohd Yusuf for making this project possible.

REFERENCES

Alexander, C. (1979). *The Timeless Way of Building*. Oxford: Oxford University Press.

Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., King, I.F. and Angel, S. (1977). *A Pattern Language: Towns, Buildings, Construction.* Oxford:Oxford University Press.

Amnon, H.E., Amiram, Y. and Joseph (Yossi), G. (1997). Precise Specification and Automatic Application of Design Pattern, in *Proceedings of the 12th IEEE International Automated Software Engineering Conference* (ASE) 1997, Nov-2-5, Lake Tahoe, CA, USA, IEE KS Press.

Appleton, B. (1997). Patterns and Software: Essential Concepts and Terminology, *Object Magazine Online Vol. 3 No 5* , pp 20-25.

Bosch, J. (1998). Design Patterns as Language Constructs. *Journal of Object-Oriented Programming, Vol. 11 No. 2,* pp. 18-32.

Brooks Jr., F.P. (1987). No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer, Vol. 20 No. 4*, pp. 10-19.

Coplien, J.O. (1998). Software Design Patterns: Common Questions and Answers, in Rising L. (Ed.), *The Patterns Handbook: Techniques, Strategies, and Applications.* Cambridge: Cambridge University Press. pp. 311-320.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Publishing Company.

Idris, N., Zamli, K.Z., Wan Hassan, W.A. and Mohd Zainuddin, N.M. (2000). UML Design Patterns for Developing Large Client Server Windows Applications, in *Proceedings of the International Wireless Telecommunication Symposium (IWTS 2000).* Shah Alam: Universiti Teknologi Mara.

Liberty, J. (1998). *Beginning Object Oriented Analysis and Design With C++.* Birmingham, UK: Wrox Press Ltd.

Muller, P.A. (1997). *Instant UML.* Brimingham, UK: Wrox Press Ltd.

Rising, L. (1996). Design Patterns: Elements of Reusable Architectures. *Annual Review of Communications, vol. 49.* pp. 907-909

Royce, W. (1998). *Software Project Management- A Unified Framework.* Boston, MA, USA: Addison-Wesley, 1998.

Rumbaugh, J., Jacobson, I. and Booch, G. (1999). *The UML Reference Manual.* Boston, MA, USA: Addison Wesley.

Voas, J. (1999). Software Quality's Eight Greatest Myths. *IEEE Software September/October 199.9* pp. 118-120.

Zamli, K.Z, Wan Hassan, W.A. and Mohd Zainuddin, N.M. (1999a). Software Development Plan (SDP) for Financial Analysis Module, Technical Report produced for Consolidated Cable (M) Sdn. Bhd and CASE, Universiti Teknologi Malaysia, January 1999.

Zamli, K.Z., Wan Hassan, W.A. and Mohd Zainuddin, N.M. (1999b). Interface Requirement Specification (IRS) for Financial Analysis Module, Technical Report produced for Consolidated Cable (M) Sdn. Bhd and CASE, Universiti Teknologi Malaysia, February 1999.

Zamli, K.Z., Wan Hassan, W.A. and Mohd Zainuddin, N.M (1999c). Software Requirement Specification (SRS) for Financial Analysis Module, Technical Report produced for Consolidated Cable (M) Sdn. Bhd and CASE, Universiti Teknologi Malaysia, April 1999

Zamli, K.Z., Wan Hassan, W.A. and Mohd Zainuddin, N.M. (1999d). Software Design Documents (SDD) for Financial Analysis Module, Technical Report produced for Consolidated Cable (M) Sdn. Bhd and CASE, Universiti Teknologi Malaysia, October 1999.

Zamli, K.Z., Wan Hassan, W.A. and Mohd Zainuddin, N.M. (1999e) Development of Financial Analysis Module for ERP Application, Technical Report for Consolidated Cable (M) Sdn. Bhd and CASE, Universiti Teknologi Malaysia, December 1999.