Application of MVP Architecture in Reengineering of Legacy Financial System

WU Bo
College of Computer Science and Technology
Zhejiang University
Hangzhou, China
bzbz@zju.edu.cn

YANG Xiaohu
College of Computer Science and Technology
Zhejiang University
Hangzhou, China
yangxh@zju.edu.cn

***Abstract*—As the progress of information technology, some financial systems used for long time can no longer meet the requirement of customers in both user interface and business process, and need reengineering imminently. In order to save the human resource and financial effort, this article raises a reengineering approach by encapsulating the back-end data access code of legacy system and only refactoring the front-end user interface and business logic. In the reengineering work, this article uses MVP (Model-View-Presenter) Architecture. By extracting screen logic into Presenters, the display code in View is independent from logic code in Presenter. Also, the introduction of MVP Architecture greatly enhances the reusability and the testability of business logic code, and finally improves the efficiency of development and test work significantly.**

Keywords- MVP Architecture, Reengineering on Legacy System, Windows Presentation Foundation, Code Reuse

I. INTRODUCTION

As the progress of information technology, some legacy financial system can no longer meet the requirement of customers in both user interface (UI) and business process. Financial companies such as bank and insurance company have strong dependency on existing legacy financial systems, as the result, reengineering and update is an imminent task for legacy financial systems. But legacy financial systems are complex for reengineering. Some systems were developed long time ago, some have few comments and documents, some are developed by out-of-time technologies, and some have very complex screen and business logics and very large amount of code.

There are multiple solutions for the reengineering of legacy system in academe and industry. Harry M. Sneed raised a five-step process based on the analysis on legacy system[1]. Vijay K. Madisetti mainly focus on the reengineering of legacy real-time systems[2]. Ettore Merlo raised a solution of evolving the old interfaces of legacy system into new ones, in order to lengthen the life-cycle of legacy system[3]. Yan Liu has done lots of research work on encapsulating legacy system into Web Services[4]. Zhan Jianfeng from Chinese Academy of Sciences raised a Mobile Agent based reengineering solution for legacy system[5]. But these solutions may cost lots of time and human resources, which is unbearable in time or in cost.

This article raise a reengineering solution of re-writing the frontend UI and encapsulating the backend logic of legacy system, based on the application of Model-View-Presenter Architecture (MVP) firstly raised by IBM[6]. This solution is adopted in the reengineering work of an actual financial system, and according to the statistical data of this program, it can save human resource and shorten the development cycle greatly. Section II will demonstrate a typical legacy financial system. Section III will focus on the theories of MVP Architecture and its difference from MVC Architecture. Section IV will mainly expatiate the specific reengineering work on legacy financial system, and reveal the improvement of this solution on both resource and efficiency. Section V will be the conclusion part.

II. A TYPICAL LEGACY FINANCIAL SYSTEM

There are varieties of investments (such as Common Stock, Bond, Option, Portfolio and so on) and various operations (such as Purchase, Sale, Stock Split, Redemption and so on) within a financial company, and an effective and overall control on such investments is quite necessary. The legacy financial system will be reengineered in this article is a Portfolio Management System (Figure 1). With the use of this system, financial company can effectively manage various securities, classify them into different portfolios, record almost all kinds of financial transactions and print reports.

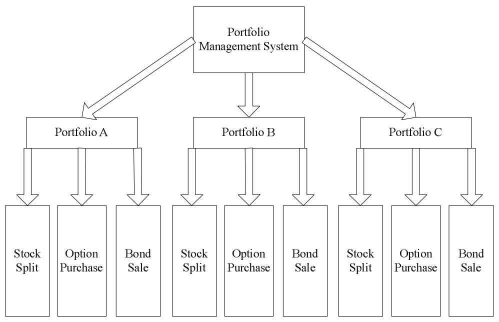

Figure 1. Portfolio Management System

Nevertheless, this Portfolio Management System is developed using Win32 API and MFC Framework in the early 1990s, and can no longer satisfy the requirement of current customers, operationally and aesthetically. Although the system still has the most powerful function and best efficiency, customers begin to complain the out-of-time UI and some hard-to-use operations, such as too many pop-up windows, no support for concurrent transaction screens and so on. All these problems need to be solved by software reengineering work.

However, this legacy system is developed about 20 years ago, with over 15 million lines of unmanaged C++ source code in total. The source code has different coding styles, few comments and almost no documents, as the result, the reengineering of this system is an arduous work.

III. MVP Architecture

When talking about MVP (Model-View-Presenter) Architecture, we cannot disregard its sibling MVC (Model-View-Controller) Architecture. They are similar but actually different. In MVC Architecture (Figure 2), Model is used to maintain business behaviors and states; View gets data from Model, demonstrates it to the user after a series of business logic, and also takes charge of the UI responding logics to user operations; Controller is playing a role of message dispatcher, sending View request and Model notification, just like a bridge connecting View and Model. In such MVC Architecture, UI responding logics are located in different Views, mixed together with UI display code. This mix-up of logic code and display code will make the UI responding logics difficult to maintain, and almost impossible to reuse.

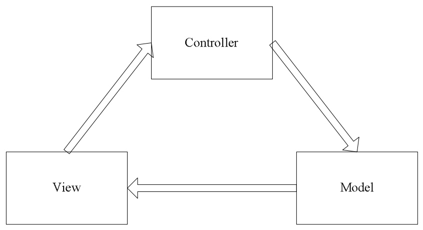

Figure 2. MVC Architecture

In contrast, MVP Architecture (Figure 3) uses Presenter to get data from Model, processes business logic within the Presenter, and displays the result on View. Nevertheless, the UI responding logics such as data selection and data classification, which is processed by View in MVC Architecture, will be extracted to Presenter in MVP Architecture. In MVP Architecture, View will only take charge of UI demonstration, by loose coupling to its corresponding Presenter.

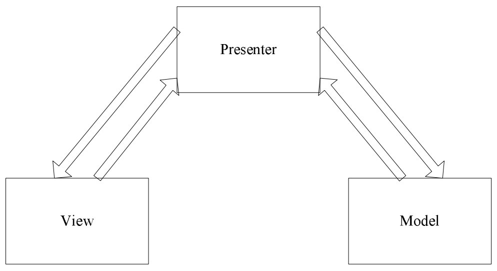

Figure 3. MVP Architecture

The application of MVP Architecture in our legacy system reengineering work can bring two significant advantages:

First, MVP Architecture extracts logic code from View into Presenter, and makes View care about only display effects. The separation of display code and logic code can make the system architecture more clear and easy to maintain. For example, developers can change the display code independently or even re-write a complete new View, just by binding it to the existing Presenter. Moreover, display code in View can be developed independently from logic code and plays the role of inter-media between developers and designers. This will guarantee the UI requirement to be executed as correctly as possible.

Second, MVP Architecture enhances the reusability of UI responding logic, and makes the UI responding logic easy to test[7]. The UI responding logic can be reused after encapsulation into Presenters, and different Views can bind to such common Presenters respectively to get the display data. Besides, UI responding logics can be tested by unit test rather than manual test from UI, since all of them are collected to Presenter.

Table 1 lists the prefered situation for MVP and MVC.

TABLE I. MVP or MVC?

Archite cture	**Better for using**
MVC	Web Application with multiple and frequently-redirecting screens.
MVP	"Fat-Client", Win Form Application with complex screen logics but fewer screen redirecting.

Based on the previous concerns, MVP Architecture is more suitable for the reengineering work of legacy financial system with complex screen logic but few screen re-directions.

IV. Case Analysis

A. *Reengineering solution for the system*

The Portfolio Management System mentioned in Section II has the architecture in Figure 4. There are mainly 3 kind of functional code: DB Access code, Business Object code and UI responding logic code. All of these codes are written by unmanaged C++, which gives the system an excellent performance. The frontend UI are developed with Win32 API and early MFC framework, out of time both aesthetically and operationally. As the result, the reengineering for frontend UI of the Portfolio Management System is the most critical work.

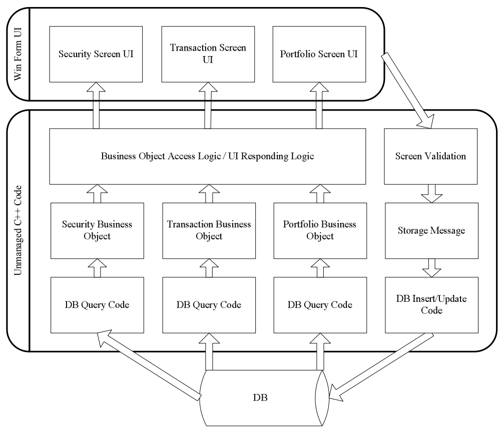

Figure 4. Architecture of Legacy System

We have the following concerns in the reengineering work:

First, the most critical problem for our Portfolio Management System lies in the UI part: the eliminating UI technology, the complexity and inconveniency for user operation, and together with the aesthetical concerns. However, the efficiency and performance of this system is still industry leading, so it's not necessary to re-write the whole system.

Second, the amount of source code for the whole system is so enormous that re-writing the whole system will cost over 3,000 man-months in total. Such expense will be unbearable economically or technically. In contrast, the reengineering solution raised in this article, which will encapsulate the backend logics and only re-write the frontend UI, will cost 1,500 man-months in total. Obviously our solution can save much more financial and human resources.

Based on the previous concerns, the reengineering solution for our legacy system is shown in Figure 5: First, use managed C++ code to encapsulate the DB Access code and Business Object code written by unmanaged C++ in legacy system, and then expose such Business Object as services through WCF (Windows Communication Foundation). Data extracted from such services can be invoked by C# code directly. Afterward, use C# code to construct different Presenters corresponding to different screens. Presenters will get Business Object data from WCF services, and contain the re-written UI responding logics. Finally, use WPF (Windows Presentation Foundation) to re-write the UI and bind each new View to its corresponding Presenters.

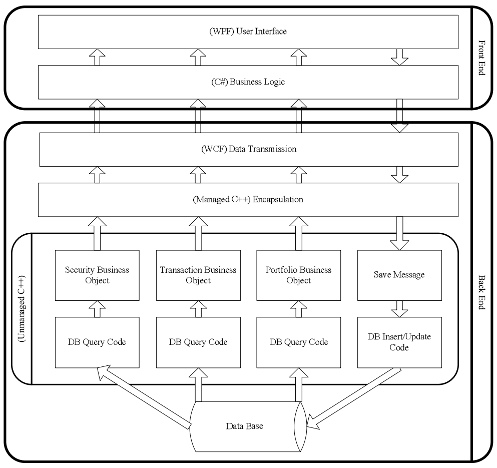

Figure 5. New Architecture after Reengineering

B. *The application of MVP Architecture*

In this article, we use MVP Architecture mentioned in Section III to implement the new frontend UI. We will use a simplified case in the following of this article, to illuminate the specific design and implementation of our new frontend UI. In this case, we are reengineering on a Stock Purchase screen, which contains only four controls: Trade Date, Settlement Date, Cancel Button and Post Button. Figure 6 is our new architecture for this screen after introducing MVP.

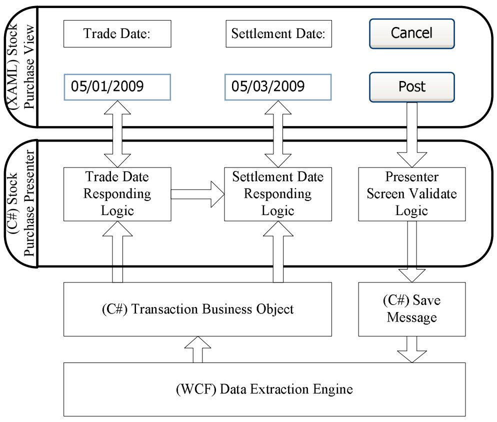
(XAML) Stock Purchase View
Trade Date:
Settlement Date:
Cancel
05/01/2009
05/03/2009
Post
(C#) Stock Purchase Presenter
Trade Date Responding Logic
Settlement Date Responding Logic
Presenter Screen Validate Logic
(C#) Transaction Business Object
(C#) Save Message
(WCF) Data Extraction Engine

Figure 6. A Simple MVP-based UI Design

From the figure above we can see, data stored in Model will come from backend DB Access and Business Object code, all of which have been exposed through WCF services. For example, backend code gets the Trade Date value from DB, encapsulates it into purchase transaction Business Object, and exposes this Business Object as WCF services. Model in frontend can get the Business Object by invoking the corresponding WCF service, and the Purchase Presenter can simply access and process the Trade Date value. WCF plays the role of a bridge, connecting the frontend C# code and backend C++ code seamlessly.

Presenter is the core of MVP Architecture, getting data for View from Model and in charge of UI responding logics. From the screen in Figure 6 we can see, the Stock Purchase Presenter, corresponding to Stock Purchase screen, constructed Trade Date Responding Logic and Settlement Date Responding Logic. Such responding logics will not only get date values from Model, but also respond to user operations. For example, Settlement Date will be calculated automatically when user changed the Trade Date. Presenter will also contain Screen Validate Logics, which will insure the validity of UI data inputted by user, after Post Button clicked.

View only contains XAML (eXtensible Application Markup Language) code, WPF provide Data Context to bind the control values on XAML with logics in Presenters. Let's keep using the example in Figure 6: Trade Date and Settlement Date from View are bound to Trade Date Responding Logic and Settlement Date Logic respectively. This binding is two-way, so modification on View will notify logics in Presenters and data processed by Presenters will display on View automatically.

In the simplified screen from Figure 6, Model is responsible for getting data from WCF services; View developed by XAML is bound to Presenter logics, and contains only display code; Screen logics such as UI responding logic and validation logic are all implemented by Presenter. Such architecture classifies code into independent parts, and makes the code easy to maintain. Moreover, the reusability and testability of the new code in MVP Architecture is greatly improved, Section 4.3 and 4.4 will focus on them respectively.

C. Common Control in MVP Architecture

Portfolio Management System has different screens for different financial transactions, as the result, we have constructed different Presenters to encapsulate their logics. However, there are some controls from different screens with the same logic. For example, calculating Settlement Date from Trade Date, selecting default bank account for a certain portfolio, loading FX rate according to currency and date and so on, all of these common logics are reusable. Lots of time and effort will be wasted, if common logics are developed repeatedly in different Presenters. To solve this problem, we introduce the concept of Sub Presenter and Sub Presenter Logic (Figure 7). Common controls implemented by Sub Presenters and Sub Presenter Logics can be used by different screen Presenters.

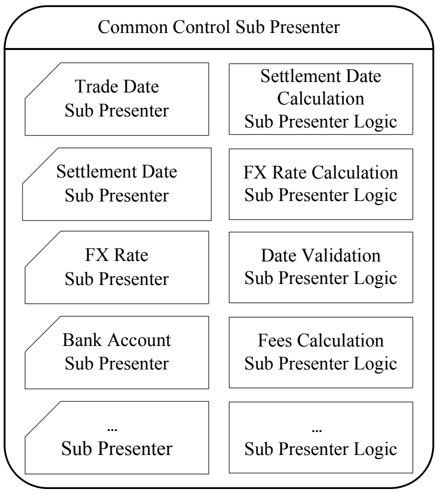
Common Control Sub Presenter
Trade Date Sub Presenter
Settlement Date Calculation Sub Presenter Logic
Settlement Date Sub Presenter
FX Rate Calculation Sub Presenter Logic
FX Rate Sub Presenter
Date Validation Sub Presenter Logic
Bank Account Sub Presenter
Fees Calculation Sub Presenter Logic
... Sub Presenter
... Sub Presenter Logic

Figure 7. Common Sub Presenter and Logic

Figure 7 shows that Sub Presenters encapsulate controls such as Trade Date, Bank Account, FX Rate, and Sub Presenter Logics encapsulate the UI responding logics and validate logics for such controls. Take the Stock Purchase screen from Figure 6 as our example, its corresponding Presenter (Figure 8) will be implemented by the following steps:

First, Construct a Stock Purchase Base Presenter, this Presenter is corresponding to the whole Stock Purchase screen.

Second, Create a Sub Presenter Manager in Stock Purchase Base Presenter. The Sub Presenter Manager is used to manage the 2 common controls within the screen, Trade Date Sub Presenter and Settlement Date Sub Presenter.

Finally, Sub Presenter Manager from Stock Purchase Screen will create Trade Date Sub Presenter and Settlement Date Sub Presenter, and attach Settlement Date Calculation Sub Presenter Logic and Date Validation Sub Presenter Logic to them each.

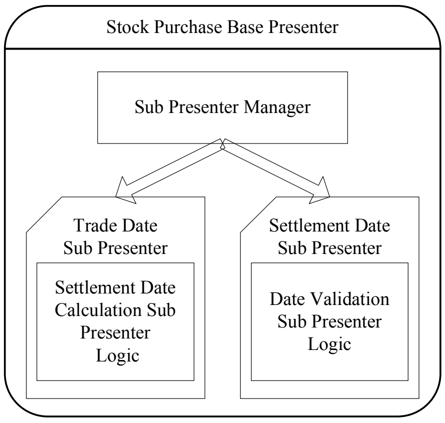
Stock Purchase Base Presenter
Sub Presenter Manager
Trade Date Sub Presenter
Settlement Date Calculation Sub Presenter Logic
Settlement Date Sub Presenter
Date Validation Sub Presenter Logic

Figure 8. The Implement of Stock Purchase Presenter

Sub Presenter and Sub Presenter Logic modularize screen logics, and make the reuse of screen logic among different Presenters possible. Sub Presenter and Sub Presenter Logic can make the screen logic code easy to maintain and reuse, and finally improve the efficiency of development and maintenance.

D. Screen Logic Test in MVP Architecture

There is very large amount of manual UI test work for system with complex screen logic such as financial systems. However, in traditional architecture such as MVC, UI display code is always messed up with screen logic code, as the result, the test for screen logic can only rely on manual UI test and automatic script test tools. For complex screens in our Portfolio Management System, such UI test has very low efficiency and may cost lots of human effort.

In contrast, MVP Architecture extracts screen logics from View into Presenter, in the form of Sub Presenters and Sub Presenter Logics. Developers can use unit test tools supplied by Visual Studio to test such screen logic code effectively.

For the Stock Purchase screen in Figure 6, we can see the differences between two test methods mentioned above from Figure 9. For example, when we are testing the logic of calculating Settlement Date after modifying Trade Date, traditional manual UI test can only change the Trade Date from screen manually and check the value change of Settlement Date. But in MVP Architecture, we need only do sufficient unit test for the Settlement Date Calculation Sub Presenter Logic, and this will insure the correctness of screen logics and mitigate the manual test work significantly.

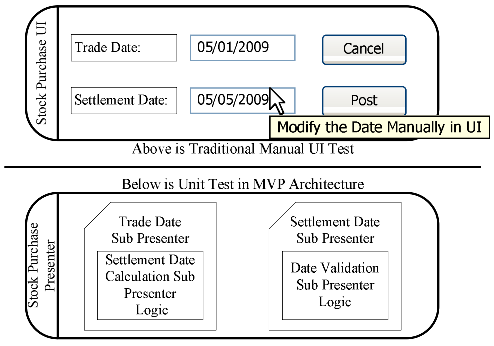
Stock Purchase UI
Trade Date:
05/01/2009
Cancel
Settlement Date:
05/05/2009
Post
Modify the Date Manually in UI
Above is Traditional Manual UI Test
Below is Unit Test in MVP Architecture
Stock Purchase Presenter
Trade Date Sub Presenter
Settlement Date Calculation Sub Presenter Logic
Settlement Date Sub Presenter
Date Validation Sub Presenter Logic

Figure 9. Traditional Manual UI Test and Unit Test in MVP Architecture

E. Evaluation for our Reengineering Solution

We entitle the solution of re-writing the whole system to Solution A, and the solution adopted in this article to Solution B. From Figure 10 we can see the difference on Man-Month effort between these two solutions.

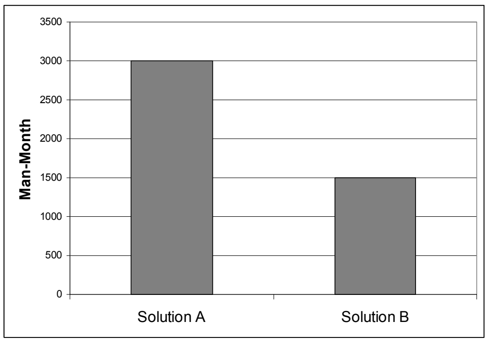

Figure 10. Man-Month Comparison for Reengineering Solution

The introduction of Common Control Sub Presenter greatly enhanced the reusability of code and development efficiency. From Figure 11 we can see that Common Controls take a part of 35% of controls from all screen Presenters. Implementation of such Common Control Sub Presenters significantly shortened the development cycle.

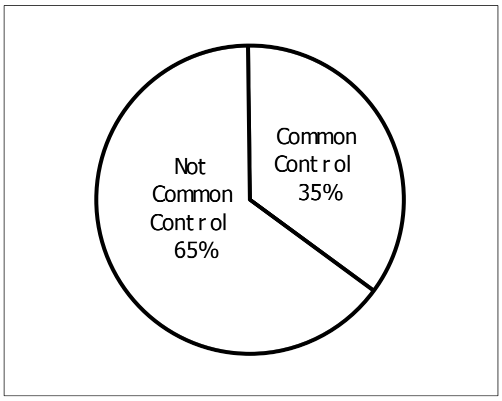

Figure 11. The Percentage of Common Control in All Controls

In MVP Architecture, developers can effectively finish unit test for screen logic code extracted in Presenter, and mitigated the manual UI test work. Figure 12 shows that the screen logic code which can be tested by unit test takes a part of 70% of all screen logics. The manual UI test work for this part of screen logic can be eliminated after sufficient unit test.

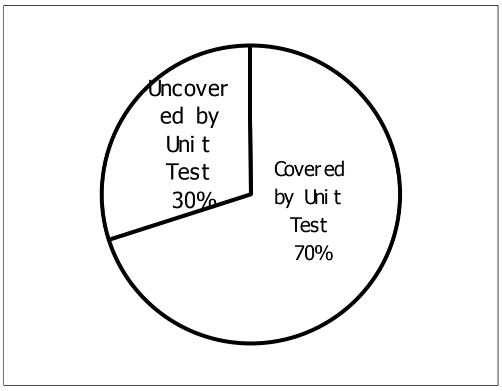

Figure 12. The Percentage of Unit Test Coverage in All Test Cases

From the evaluation result we can see that the solution adopted in this article effectively limited the human and financial resource for development, and shortened the cycle for both development and test work.

V. CONCLUSIONS

The reengineering solution raised in this article, by encapsulating the backend code and re-writing frontend code with MVP Architecture, can greatly save the financial expense and human resource for reengineering of legacy financial system. The application of MVP Architecture in the reengineering of system with complex screen logics successfully achieves the separation of logic code from display code. Moreover, by using Sub Presenter to implement Common Controls, the reusability of the code and the efficiency of development are greatly enhanced. And finally, by extracting the screen logics from Views to Presenters, MVP Architecture makes the screen logic code adaptable to unit test, which can significantly improve the efficiency of test work.

REFERENCES

[1] Harry M. Sneed, "Planning the Reengineering of Legacy System[J]", IEEESoftware. 1995: (1): 24－34.

[2] Vijay K. Madisetti. Yong－KyuJung. Moinul H. Khan, "Reengineering Legacy Embedded Systems[J]", IEEE Designand Test of Computers, 16(2): 38－47, 1999.

[3] EttoreMerlo, Pierre － YvesGagne, Jean － Francois Girard, "Reengineering User Interfaces [J]", IEEESoftware, 12(1): 64－73, 1995.

[4] Yan Liu, Qingling Wang, Mingguang Zhuang, Yunyun Zhu,"Reengineering Legacy Systems with RESTful Web Service [C] ", In proceeding of 32rd International Computer Software and Applications Conference, 2008.

[5] Zhan Jianfeng, Cheng Hu, "A Solution to Reengineering the Legacy System Based on Mobile Agent Technology[J]", Journal of Software, 13（12）：2343－2348, 2002.

[6] Mike Potel, "MVP: Model-View-Presenter. The Taligent Programming Model for C++ and Java", 1996.

[7] Martin Fowler, "GUI Architectures", 2006.