Re-engineering a Credit Card Authorization System for Maintainability and Reusability of Components – *a Case Study*

Kyo Chul Kang[1], Jae Joon Lee[1*], Byungkil Kim[1], Moonzoo Kim[1], Chang-woo Seo[2], and Seung-lyeol Yu[2]

[1] Software Engineering Lab. Computer Science and Engineering Dept.
Pohang University of Science and Technology, Pohang, South Korea
{kck,gibman,dayfly,moonzoo}@postech.ac.kr

[2] System Development Team and Quality Management Team, LG-Card Co.
118, Namdaemun 2 ga, Jung-gu, Seoul, South Korea
{cwseo, toto}@card.lg.co.kr

Abstract. A credit card authorization system (CAS) is a large information system performing diverse activities such as purchase authentication, balance transfer, cash advances, etc. One characteristic of CAS is its frequent update to satisfy the needs of customers and newly enforced governmental laws. Thus, CAS should be designed to minimize the effects of updates, for which high reusability of the CAS components is desired. In this paper, we present our experience of re-engineering CAS based on a feature model for improved reusability of components, which alleviates the difficulty of system maintenance. The result of this project has been successfully transferred to the company.

1. Introduction

A credit card authorization system (CAS) is one of the largest information systems used worldwide. CAS handles various types of transactions in large volume, such as purchase authentication, balance transfer, affiliated discount services, etc. One characteristic of CAS is its frequent update, and the maintainability of CAS is a crucial issue for credit card companies. Government frequently creates and enforces laws targeting the business of card companies. In addition, due to heavy competition in the credit card market, card companies are pressed to offer new services or change existing services frequently. For example, the discount rate on gas purchase for freight vehicles changes many times a year due to gas price changes and discount rate changes of other card companies. These situations cause constant revisions of CAS, which increases the complexity of system maintenance. Thus, in order to manage

[*] Jae Joon Lee is currently at Fraunhofer Institute for Experimental Software Engineering, Fraunhofer-Platz 1, D-67663 Kaiserslautern, Germany (jaejoon.lee@iese.fraunhofer.de)

frequent revisions, CAS should be designed to accommodate changing requirements easily and isolate effects of updates as much as possible.

From the review of the CAS of LG Card Co. Ltd, we found several opportunities to enhance reusability of the CAS components. One manifest problem was that new services have been added to CAS by simply adding new components specially developed for those services without consideration of common/reusable characteristics of the services. This was caused by the lack of *proactive design* that anticipates updates of services based on market evolution. This ad-hoc way of evolution resulted in redundant code and difficulty of understanding program behavior. As a result, newly added services or updates of services easily affected unnecessarily large segments of CAS and caused high maintenance costs.

In this paper, we present our experience of improving reusability of the CAS components through *proactive re-engineering* based on a feature model. First, we reviewed the existing CAS code and the revision history with help of domain experts and extracted the legacy design. Then, we constructed a feature model of the CAS domain that captures variabilities of CAS from the revision history and a market analysis [1][2][3]. Based on the recovered legacy design and the feature model, we could re-design components of CAS to preplan adoption of future evolution, which enhanced system maintainability. This re-engineering task was conducted based on three re-engineering principles: encapsulation of variabilities, generalization of common processes, and separation of data-streams.

Section 2 describes related works briefly. Section 3 gives an overview of CAS and its corresponding feature model. Section 4 explains the three design principles we applied to the re-engineering task. In section 5, we explain details of the re-engineering task. Lessons learned from this re-engineering project are summarized in section 6. Finally, we conclude this paper with future works in section 7.

2. Related Work

There have been active researches for improving maintainability and reusability of software systems. One of difficult problems in software maintenance is that there exists duplicated code among multiple components that enlarges change efforts and, thus, increases the difficulty of maintaining systems. In order to alleviate this problem, in addition to applying fundamental software engineering principles such as decreasing component coupling and increasing functional cohesion [4], software metric [5], software visualization [6], and concept analysis [7] have been used.

There are several important classes of researches focused on reusability in the information systems domain. For information systems, process workflows should be designed with consideration of reusability of the components that handle business processes. The workflow management coalition [8] defines a standard architecture and component interfaces to design workflows conveniently. [9] studies reuse of existing workflows based on the characteristics of data dependency among processes. [10] proposes guidelines for architecture design and development processes for reusable business components, and [11] describes refactoring techniques focusing on improved system maintainability.

Although these works contribute to enhancing reusability and maintainability of business components and workflows, domain analysis to encapsulate variabilities and reuse commonalities must precede these activities in order to enhance the benefits further. We use a feature model for domain analysis, and apply the analysis results and re-engineering principles to make a proactive design for improved reusability and maintainability.

3. Overview of Card Authorization System

This section describes the background of the CAS domain. We explain the background of this project in section 3.1 followed by an overview of CAS in section 3.2. The feature model of CAS is given in section 3.3.

3.1 Background of the Re-engineering Project

In the year 2004, LG-Card Co. Ltd [12] adopted a component based development (CBD) method [13][14] and started to re-develop CAS by converting hard-coded business rules into a database and standardizing component interfaces. Moreover, to enhance the reusability of components, they continuously applied several component based management (CBM) programs [15][16] such as reuse rate measurement, component library construction, and component re-engineering. Nevertheless, they had difficulties in maintaining CAS. The developers added/updated components in an ad-hoc way at each update request, which brought about duplicated code and complex component interactions. As a result, this reactive maintenance caused high maintenance costs even with simple changes.

To solve these problems, LG-Card requested POSTECH in the year 2005 to evaluate and improve the *credit card* and *check card systems* that are the core of the entire CAS. With the request, the POSTECH team studied the CAS domain and re-engineered CAS for six months to enhance maintainability by improving the reusability of CAS components.

3.2 Overview of CAS

Fig.1 shows an overview of CAS. The left part of Fig.1 shows CAS and its environment. CAS interacts with NET24, a middleware working as an interface between CAS and banks, point of sales (POS), and customer services. In addition, CAS communicates with a database system to retrieve and update transaction information. The Net24Main component of CAS directly interacts with NET24 and distributes transaction requests from outside to the credit card system or to the check card system accordingly. Each card system consists of four component layers: transaction classifier (TC), transaction flow manager (TFM), business process component (BPC), and interface component (IC).

The main task of TC is to classify transaction types and to call appropriate TFM components. The TFM components manage transaction flows by controlling business processes implemented in the BPC components. The IC components work mostly as data holders communicating with the database system. The component manager handles orderly creation of these components preventing redundant instantiation. Components of a higher layer control the components of a lower layer via call/return methods; a TC component calls appropriate TFM components, then a TFM component calls BPC components, etc.

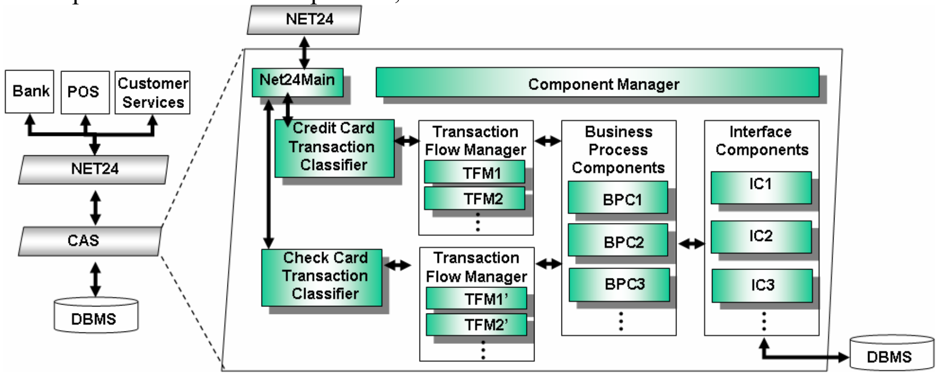

Figure 1 – Overview of CAS

Fig.2 illustrates how these layers work in the check card system.[1] When a user purchases a product using his/her check card, a purchase authorization request is sent from the store to CAS. Then, ChkCdClsf, a TC component for classifying check card transactions, recognizes the type of the transaction and calls VrfyReqTrs, a TFM component, to check if the requested transaction is valid or not. VrfyReqTrs calls TransVldChk, ScrUserAuth, CdInfoChk, and others in sequence. TransVldChk identifies the place where the transaction occurs. If the transaction occurs at an online store, ScrUserAuth is called to check the user's identification and password. Otherwise, ScrUserAuth is not invoked. Then, CdInfoChk is called to verify whether the given card information, such as the expiration date and the name of cardholder, matches with the information in the CAS database. Similarly, other BPC components are also called according to the transaction flow encoded in VrfyReqTrs. Once VrfyReqTrs has finished authorization of the transaction, VrfyReqTrs sends a request to the bank that issued the check card to check if the bank account has enough balance for the purchase. After receiving that request, the bank sends to CAS a new transaction request that contains required information. Then, the request is passed to RespTrs, and RespTrs calls TransVldChk.[2]

[1] The credit card system has 4 TFM components and the check card system has 5 TFM components. There exist 39 BPC components and 71 IC components shared by both systems.

[2] Note that VrfyReqTrs and RespTrs share many BPC components because these two TFM components perform similar tasks. This redundancy problem comes from the legacy design of earlier CAS and has not been fixed yet due to risk of a large scale revision.

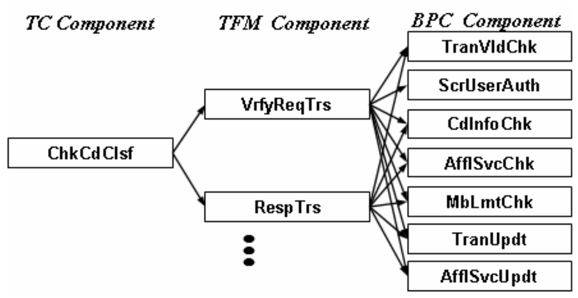

Figure 2 – Execution flow of the check card transactions

3.3 A Feature Model of CAS

In order to improve the maintainability of CAS, we need to re-design CAS with consideration of potential service changes based on the revision history and market projections. A feature model is a suitable tool to capture variability of services and make a proactive design for evolving services based on relationships among features. We analyzed commonalities and variabilities of the CAS domain first and developed a feature model shown in Fig. 3.[3] CAS (the root node of Fig.3) consists of "Check Card Authentication" and "Credit Card Authentication" each of which corresponds to authentication features for check card and credit card respectively. In addition, CAS has "*Affiliated Service*" feature that represents various affiliated services such as purchase discount, free service, etc.

Features that change frequently are named in italic font in Fig. 3. For example, features related to "*Affiliated Service*" are indicated as frequently updated features. Table 1 includes the revision history of CAS from July 2005 to August 2005. As can be seen in Table 1, there are frequent revisions due to newly added affiliated stores and changes of affiliated services. "*Discount Service*" (located at the top right corner of Fig.3) is an affiliated service for handling purchase discounts. This feature is specialized to "*Handicapped Welfare Service*" and "*Freight Car Oil Supp. Service*" features. "*Handicapped Welfare Service*" is a service that provides discount when a handicapped cardholder purchases daily necessities. "*Freight Car Oil Supp. Service*" provides discount to a freight vehicle driver for gas purchases. Changes of "*Discount Service*" affect "*Discount Service Update*", "*Discount Service Check*", and "*Discount Limit Check*" as indicated by the configuration dependency relationships.

As shown in Fig.3, changes in one feature (e.g., "*Discount Service*") can affect several other features (e.g., "*Discount Service Update*" and "*Discount Service Check*"), which requires to modify related components altogether. This is a complex task without knowing explicit relationships between features. After we identified frequently changing features and related components, we could re-engineer these components to prepare adoption of future revisions by improving reusability of these components.

[3] Features corresponding to business processes are not shown in this paper.

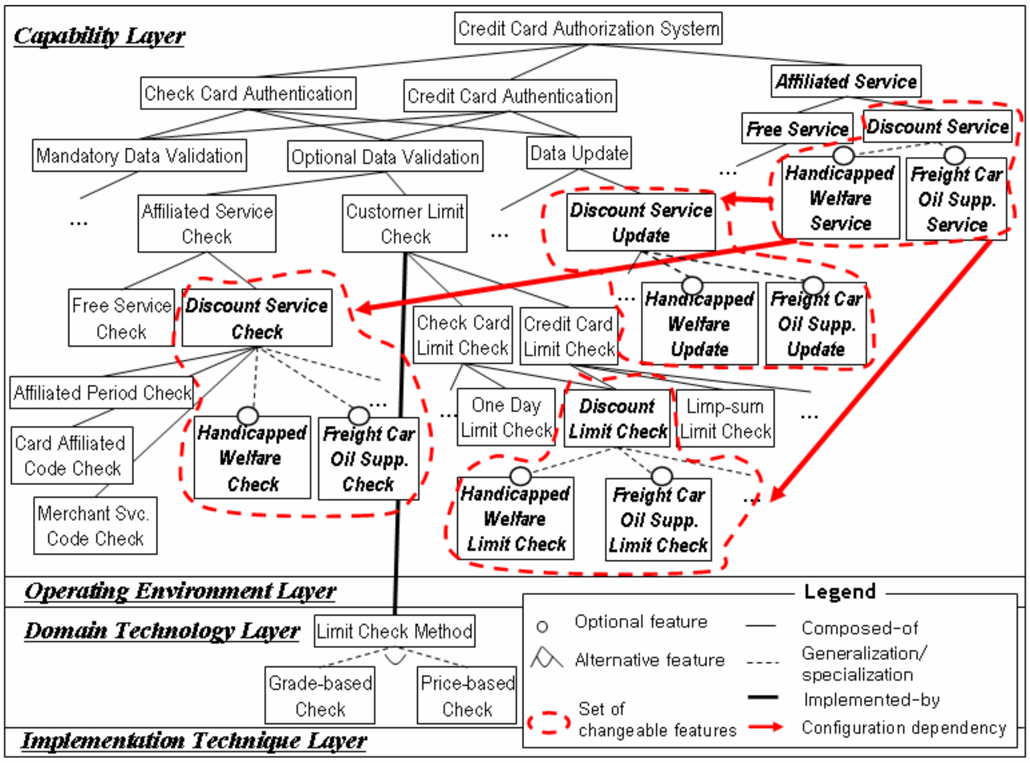

Figure 3 – A feature model of CAS

Date	Revisions
Aug 16	Changed codes of refusing transactions for a family restaurant discount service. - If the service is not applicable to the card owner, return the refusal code 588. - If there was no transaction in the previous month, return the refusal code 593.
	Added an affiliated service code for the DW department store.
Aug 4	Added an affiliated service code for the KB department store.
	Added a business process to restrict a discount service for “LG BF” Card
July 13	If a welfare service for handicapped people is requested by a handicapped user using a family card, the transaction should be refused.
	Changed a business process for the oil discount service for freight vehicles - removed freight vehicle oil discount codes K002 and K003 - modified the codes between K011 and K020.
July 11	Added an affiliated discount service for DJ Zoo.
July 04	Added a business process for a discount service used by MIC

Table 1 – CAS revisions between July 2005 and Aug 2005

4. Re-engineering Principles

Based on the review of the design and the revision history of CAS, we propose three principles for making a *proactive system design* through re-engineering. These principles can serve as a primary design plan, which is indispensable in any engineering projects of sizable scale.

4.1 Principle 1: Encapsulation of Evolving Features

A complex system like CAS usually suffers from high degree of coupling among components. This problem often occurs when new components for the requested services are implemented by copying and modifying existing components without reorganizing/refactoring them. High coupling among components makes the behavior of the system difficult to understand and, thus, it is hard to revise and maintain the system, degrading evolvability.

To address this problem, we propose to encapsulate frequently changing features in a component. In other words, we group components for evolving features into a module that provides a common interface to the rest of the system. By this encapsulation, we can decrease the degree of component coupling and localize the effects of component update into a module. In addition, details of components can be abstracted away, which provides better system understandability.

4.2 Principle 2: Generalization of Common Processes

Information systems provide a large number of services some of which are similar with minor distinction. Thus, without careful anticipation of changes, multiple components with slightly different services easily prevail in the system. When a change is made to a common process, multiple components that implement the process should be modified altogether. In addition, it becomes hard to find which component is responsible for a specific behavior of the system, which degrades the maintainability of the system.

Therefore, it is highly desirable to make generalized components for common processes so that the degree of redundancy could be decreased. In addition, sequences of processing various transactions are valuable domain knowledge that should be reused to minimize the risk of creating wrong process sequences. Once common processes are identified, we can build generalized components for the common processes and then extend the components for specialized processes using inheritance and/or association mechanisms.

4.3 Principle 3: Separation of Upstream Data from Downstream Data

As typical of information processing systems, the main operations of CAS are to retrieve, process, and update data. Thus, clear and efficient handling of data is at the core of quirements. For this purpose, all data-streams among components must be clearly defined. In other words, the source and the destination of a data-flow must be identified clearly and patterns of data-flows must be visible. This clear identification of data-streams helps preventing unnecessary modification of multiple components that access a data-stream.

One way of achieving this goal is to classify data-flows explicitly based on its characteristic. CAS has a layered architecture consisting of TC, TFM, BPC, and IC components that process transactions in order. Thus, we could identify two separate data-streams as follows:

- TC→TFM→BPC→IC for managing transaction information (*downstream*)
- IC→BPC→TFM→TC for reporting result of transaction validation (*upstream*)

Based on this information, we could separate data-streams in two directions explicitly, which provided optimized data structures for each data-stream as well as localization of change effects when an update to data handling components happened.

5. Re-engineering CAS

In this section, we describe details of the re-engineering task. In section 5.1, we show how to encapsulate BPC components based on their characteristics. Section 5.2 explains the design of generalized components. Finally, section 5.3 describes separation of an upstream data-flow from a downstream one.

5.1 Encapsulation of the BPC Components

In order to improve the reusability of CAS components, it is necessary to minimize the effects of updates as much as possible. The current layered architecture was designed to achieve this goal by embedding reusable business processes into the BPC components (i.e., components that embed business workflows are separated from the components of functional tasks) so that when a business process changes, its effects would be localized to the corresponding BPC component and the TFM components that control the BPC component directly. For example, in Fig. 2, suppose that AfflSvcUpdt BPC component is modified by changing a process of handling affiliated services. Then, all TFM components accessing AfflSvcUpdt, such as VrfyReqTrs and RespTrs, should be modified accordingly in the original design.

Considering that most business processes of frequently changing services (e.g. affiliated services) are embedded in the BPC components, it is crucial to minimize update effects of the BPC components. We noted that effects of updating the BPC components could be reduced further by encapsulating BPC components. In other words, we grouped BPC components of similar characteristics into a module to minimize the change effects. For this goal, we studied the workflow of CAS carefully and grouped BPC components based on their data usage. Fig. 4 shows workflows of business processes of CAS. The table in the left part of Fig. 4 shows what these processes are and the flowgraph in the right part of Fig.4 shows how these processes are connected and executed in order.

First, we classified the business processes according to its type of data manipulation – validation (*read*) and update (*write*). Processes 1 through 11 are to validate transaction information and processes 20 through 26 are to update validation results into the CAS database. Second, we classified processes 1 through 11 further based on the type of data. Processes 1 to 4 validate mandatory transaction data that should be validated even for simple transactions such as purchase cancellation.[4] Processes 5 to 11 validate optional data such as data about affiliated services (process 8) and short message service (process 11). Based on this classification, we grouped processes 1 through 4 into the MdDataVld module, processes 5 through 11 into the OpDataVld module, and processes 20 through 26 into the DataUpdate module.

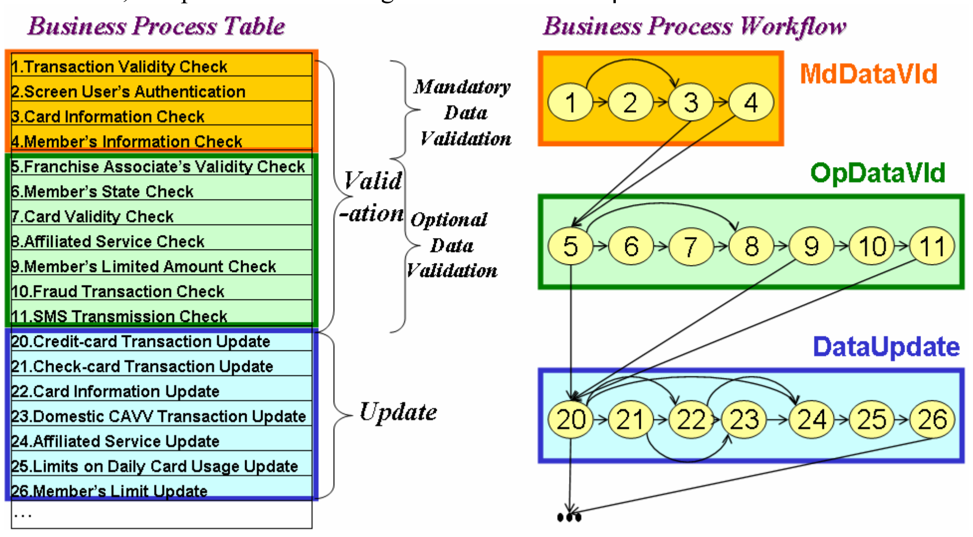
Business Process Table
1.Transaction Validity Check
2.Screen User's Authentication
3.Card Information Check
4.Member's Information Check
5.Franchise Associate's Validity Check
6.Member's State Check
7.Card Validity Check
8.Affiliated Service Check
9.Member's Limited Amount Check
10.Fraud Transaction Check
11.SMS Transmission Check
20.Credit-card Transaction Update
21.Check-card Transaction Update
22.Card Information Update
23.Domestic CAVV Transaction Update
24.Affiliated Service Update
25.Limits on Daily Card Usage Update
26.Member's Limit Update
...
Valid -ation
Mandatory Data Validation
Optional Data Validation
Update
Business Process Workflow
MdDataVld
OpDataVld
DataUpdate
1 2 3 4
5 6 7 8 9 10 11
20 21 22 23 24 25 26

Figure 4 – Workflow of the business processes of CAS

Fig. 5 shows the re-engineered component design. By encapsulating BPC components, we could localize the effect of updating a BPC component to only one module that incorporates the BPC component instead of multiple TFM components. In the original design, if AfflSvcUpdt BPC component is changed, all TFM components that directly access AfflSvcUpdt such as VrfyReqTrs and RespTrs need to be modified accordingly. In the new design, however, the effect of update is localized to the DataUpdate module only. In addition, this restructuring alleviated redundancy among the TFM components because common tasks among multiple TFM components to control BPC components were extracted into a new module. For example, task of controlling TransVldChk, ScrUserAuth, CdInfoChk is moved from TFM components, VrfyReqTrs and RespTrs, to the MdDataVid module. In the original architecture, when such a common task of TFM components is updated, we had to modify all related TFM components. In the new architecture, however, we need to modify only a corresponding module.

[4] In Fig.4, the workflow for purchase cancellation consists of processes 1-4, 5, and 20.

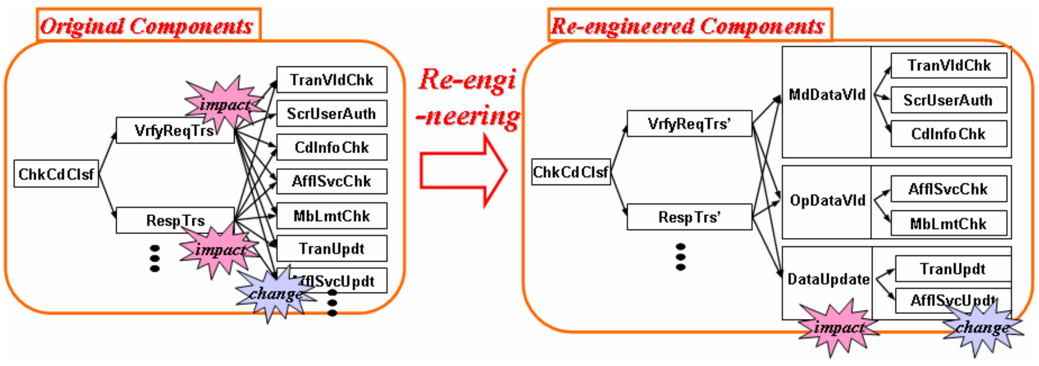

Figure 5 – Encapsulation of the CAS components

5.2 Generalization of the Common Business Processes

As can be seen in Fig. 2, the BPC components were designed to share common business processes for various transactions (even transactions of different systems – the credit card system and the check card system). Existing BPC components were, however, implemented without considering how the authorization processes could be changed. Accordingly, when new services were added, the CAS maintainer simply added new components for the services although these services could be provided with less effort by using a general component for the common processes of the services. This is a typical weakness of reactive maintenance without proactive design.

Let us look at an example. The feature model in Fig. 6 shows two sets of features for checking discount services, containing "Handicapped Welfare Check" (checking if welfare discount service is applicable) and "Freight Car Oil Supp. Check" (checking if a gas purchase discount for freight vehicles is available). These two sets of features were implemented in the HandiWelfareChk and FCarOilSuppChk BPC components respectively. These two BPC components share a same sequence of processes such as checking the affiliate service code first, then the period of affiliated service contract, then the merchant codes, etc. This fact is reflected in the feature model in Fig.6 showing that these two features are specialized instances of the "Discount Service Check" feature that contains "Card Affiliated Code Check", "Affiliated Period Check", and "Merchant Svc. Code Check" features.

Considering the fact that services are added and changed frequently (see Table 1), ad-hoc addition of components for services (e.g. FCarOilSuppChk) should be avoided because it causes redundancy among components. Thus, we need to re-engineer components so that services of common characteristics should be provided by generic components. Re-engineered components in Fig.6 show that GenDiscntSrvcChk implements common processes of both HandiWelfareChk and FCarOilSuppChk (e.g. checking the affiliated card code, affiliated period, and affiliated merchant code) by generalizing these processes. Thus, HandiWelfareChk and FCarOilSuppChk are built as extensions of GenDiscntSrvcChk.

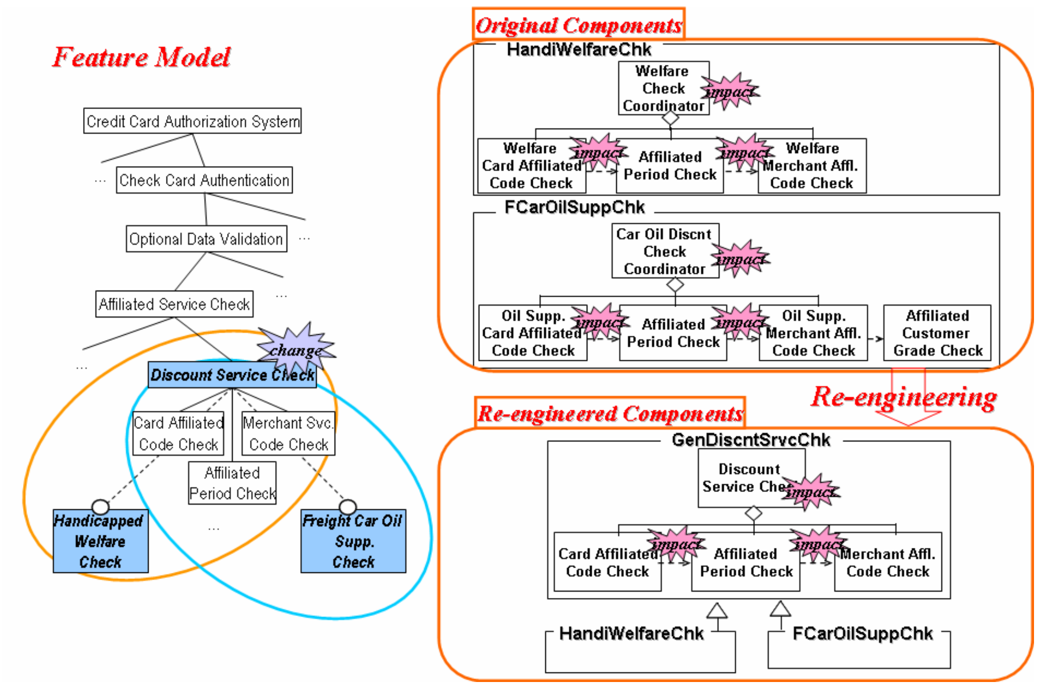
Feature Model
Credit Card Authorization System
Check Card Authentication
Optional Data Validation
Affiliated Service Check
change
Discount Service Check
Card Affiliated Code Check
Merchant Svc. Code Check
Affiliated Period Check
Handicapped Welfare Check
Freight Car Oil Supp. Check
Original Components
HandiWelfareChk
Welfare Check Coordinator
impact
Welfare Card Affiliated Code Check
Affiliated Period Check
Welfare Merchant Affl. Code Check
FCarOilSuppChk
Car Oil Discnt Check Coordinator
Oil Supp. Card Affiliated Code Check
Affiliated Period Check
Oil Supp. Merchant Affl. Code Check
Affiliated Customer Grade Check
Re-engineering
Re-engineered Components
GenDiscntSrvcChk
Discount Service Check
Card Affiliated Code Check
Affiliated Period Check
Merchant Affl. Code Check
HandiWelfareChk
FCarOilSuppChk

Figure 6 – Generalization of the CAS components

The re-engineered components have reusability benefits. Suppose that "Discount Service Check" feature is changed, e.g., the affiliated period should be checked first. With the original design, we need to update all BPC components that check discount services (e.g. HandiWelfareChk and FCarOilSuppChk). This is a burdensome job because there are many such services. In contrast, we only need to update GenDiscntSrvcChk in the new component design.

5.3 Classification of Data-streams – Upstream v.s. Downstream Data-flows

In the original design, components communicate with each other using *valued objects* (VOs), which are global data objects containing the transaction information and the result of validation. A TC component writes down transaction information into VOs and passes reference pointers to the VOs to TFM components. The TFM components read the transaction information from the VOs. Similarly, a TFM component processes the transaction data (e.g. converting a card number into a format compatible with the database), writes the processed data (e.g. converted card number) into the VOs, and then passes the reference pointers to BPC components.

This way of communication is simple but problematic. First of all, as depicted in the left part of Fig.7, data-streams among components become obscure so that it is hard to visualize component interactions; it is not clear which components modify VOs and which components are affected by the modification because data-streams are implicitly constructed through VOs. In addition, cohesion of VO is low because VO serves multiple purposes of various components. Furthermore, once a component that accesses VO is modified, all components and VO should be modified accordingly because VO works as a medium for communications of different types without hiding information. Suppose that we modify a routine of updating VO in a BPC component. Then, VO as well as other TC, TFM, BPC, and IC components can be affected altogether.

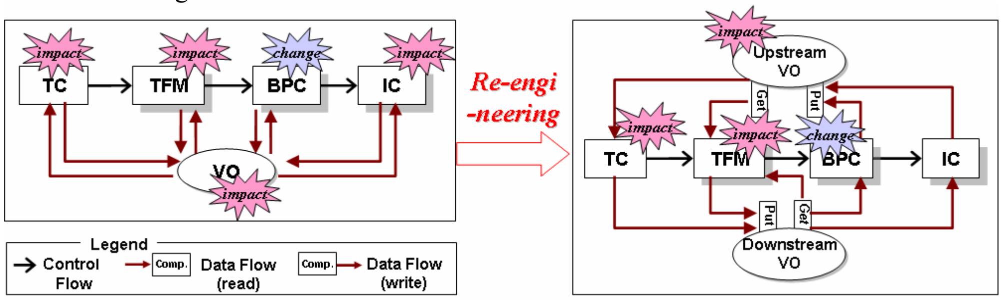

Figure 7 – Separated data-streams

We could solve these problems by separating data-streams into upstream data (containing result of transactions validation) and downstream data (containing transaction information) as depicted in the right part of Fig. 7. For downstream data, only TC and TFM components write transaction data into the downstream VOs. BPC and IC components can read from the VOs, but do not write to the VOs. For the upstream data, IC and BPC components write the result of transaction validation and TFM and TC components read the result from the upstream VOs. Furthermore, we apply the facade pattern to both upstream and downstream VOs to hide internal modification of VOs.

As a result, each data-stream could have its own VO data structure optimized for its own purpose. Also, the data structure of the upstream VO is immune from changes of the downstream VO, and vice versa. Similarly, this separation localizes the effect of modifying a BPC component to the upstream VOs and TC/TFM components. Furthermore, component interactions became visible and the responsibilities of components became clear. Thus, we could anticipate easily which components should be modified at changes of services.

6. Lessons Learned

In this section, we share the lessons learned from this re-engineering project.

6.1 Necessity of Proactive Re-engineering

During the project, we are convinced that proactive re-engineering is essential, not optional in many ways. We found several poorly designed legacy CAS components that caused high maintenance costs. Due to lack of analysis on commonality and variability of services, developers tend to revise the system in an ad-hoc manner without considering how the system should be designed for better maintainability. This way of revision results in high degree of redundancy and component coupling, which degrades maintainability of the system severely as revisions are repeated. Therefore, proactive re-engineering should be enforced to preplan efficient adoption of future evolution of services by improving reusability of components. For this re-engineering activity, a feature model works as a very effective means for capturing variability of features and creating a proactive design.

6.2 Management of Commonality and Variability

We found that the company had difficulty in managing variabilities of services systematically. One of the goals in this project was to enhance adaptability of CAS to frequently changing services. Dependency relationships between features of the feature model helped us to recognize the effects of service changes/additions. In addition, generalization/specialization relationships helped us to encapsulate similar components into generalized ones and to adopt new services more conveniently (see section 5.2). Thus, the feature model expressed valuable information for identifying both variable services and the boundary of component reuse, which supported adoption of future service evolutions. Similarly, workflow analysis helped us to understand what processes should be mandatory or optional.

6.3 Broad Coverage of a Feature Model for System Analysis

In our experience, the feature model successfully provided guidelines for analyzing the target system in a broad way, from architectural issues to component refactoring. This is because a feature model represents the domain of the target system *hierarchically*. In other words, features of higher level (near to the root of a feature tree) are related to system assets of a large scale such as an architecture or layers. In contrast, features at leaf nodes are mostly related to small objects of a system. Therefore, once a feature model is built carefully, the model can be used for analyzing a system in various levels of abstraction; the model can provide abstract views on the system domain as well as detailed views on relationships among concrete system entities.

7. Conclusion and Future Work

In this paper, we described our experience of re-engineering CAS for enhanced maintainability and reusability of components. Through the proactive re-engineering task based on the feature model, we could achieve this goal and the result was transferred to the company successfully. We believe that this case study can serve to promote the significance of proactive re-engineering based on a feature model, which can alleviate difficulties of system maintenance and reduce overall maintenance costs. As a future work, we will investigate systematic methods for validating re-engineering process, i.e., to show that re-engineered systems behave “equivalently” to the original systems.

References

1. J.Bergey, L.O’Brien, and D.Smith. Option Analysis for Re-engineering (OAR): A Method for Mining Legacy Assets (CMU/SEI-2001-TN-013). Pittsburgh, PA:Software Engineering Institute, Carnegie Mellon University (2001)
2. M.Kim, J.Lee, K.C.Kang, Y.Hong, and S.Bang. Re-engineering Software Architecture of Home Service Robots: A Case Study, International Conference on Software Engineering, Missouri, USA, pp.505-513 (2005)
3. K.C.Kang, M.Kim, J.Lee, and B.Kim. Feature-oriented Re-engineering of Legacy Systems into Product Line Assets, The 9th International Software Product Line Conference, Rennes, France, pp. 45 – 56 (2005)
4. C.Ghezzi, M.Jazayeri, D.Mandrioli. Fundamentals of Software Engineering 2nd ed., Prentice-Hall (2004)
5. B.Magdalena, M.Ettore, D.Michel, L.Bruno and K.Kostas. Measuring Clone Based Reengineering Opportunities, Sixth International Software Metrics Symposium (METRICS'99), p. 292 (1999)
6. D.E. Baburin et al. Visualization Facilities in Program Re-engineering, Programming and Computer Software Vol 27 No. 2, pp. 69-77 (2001)
7. G. Snelting and F. Tip. Re-engineering Class Hierarchies Using Concept Analysis, Proc. Foundations of Software Eng., pp. 99-110 (1998)
8. D. Holinsworth. The Workflow Reference Model, Workflow Management Coalition, TC00-1003, 1995
9. N.Russell, W.M.P. van der Aalst, A.H.M. ter Hofstede and D.Edmon. Workflow Resource Patterns: Identification, Representation and Tool Support, In the proceeding of the 17th Conference on Advanced Information System Engineering (CAiSE'05), Porto, Portugal, 2005
10. G. Baster, P. Konana, and J. E. Scott. Business Components: A Case Study of Bankers Trust Australia Limited, Communication of the ACM, Vol.44, No.5, 2001
11. C. J. Neill and B. Gill. Refactoring Reusable Business Components, IEEE Computer Society, 1520-9202/03, 2003
12. LG Card Co. Ltd homepage http://www.lgcard.com
13. D.D’sousz and A.Willi. Object, Components, and Frameworks with UML: The Catalysis Approach, Addison-Wesley (1998)

14 K.C.Kang. Issues in Component-Based Software Engineering, Proceeding of the 21st International Conference Software Engineering (1999)

15. N.Boertien, M.Steen and H.Jonkers. Evaluation of Component-Based Development Methods, International Workshop on Evaluation of Modeling Methods in Systems Analysis and Design (2001)
16. S.A.Bohner. Extending Software Change Effect Analysis into COTS Components, Proceedings of the 27th Annual NASA Goddard/IEEE Software Engineering Workshop (2003)