

# Challenge for the SPL Approach in Enterprise Software Development

**Yuzo ISHIDA** is a senior application engineer in the Distribution Systems Development Department of NRI. He is a software architect and engaged in designing and developing an application framework by integrating dozens of open source products. His specialties include mining reusable source codes from applications and building context-independent assets for the following application developments by pursuing multi-dimensional separation of concerns (MDSOC).

## 1 Background of Enterprise System Development

## 2 History of Our Challenges

- 1 Developing Semi-Made-to-Order Package Software (2001 – 2002)
- 2 Establishing an SPL Platform and Verifying it Outside of Our Domains (2003 – 2004)
- 3 Restructuring of Mission Critical Enterprise Systems on an SPL Platform (2005 – 2006)
- 4 Extending SPL Activities Further Inside and Outside Our Domains (beginning in 2007)

## 3 Design Strategies

- 1 Handling Three Different Levels of Variability Management at One Time
- 2 Applying the MDSOC Approach
- 3 Resolving a Variety of Impedance Mismatches Throughout a System

## 4 Evaluation and Lessons Learned

## 5 Conclusions

Software product lines (SPL) is a promising approach to increasing the productivity and quality of software-intensive system development dramatically, which has been proven by many case studies and reports during the last several years. However, there are very few published cases that apply the SPL approach in enterprise system development, especially on a large scale. This paper summarizes NRI's continuing challenge since 2001 to apply an SPL approach primarily in the retail industry and explains some proven architectural design strategies with lessons learned from our experiences.

**Keywords:** SPL (Software product lines), enterprise software, software reuse, variability management, MDSOC (Multi-dimensional separation of concerns), impedance mismatch, open source

# 1 Background of Enterprise System Development

The software-intensive enterprise system development usually has the following characteristics:

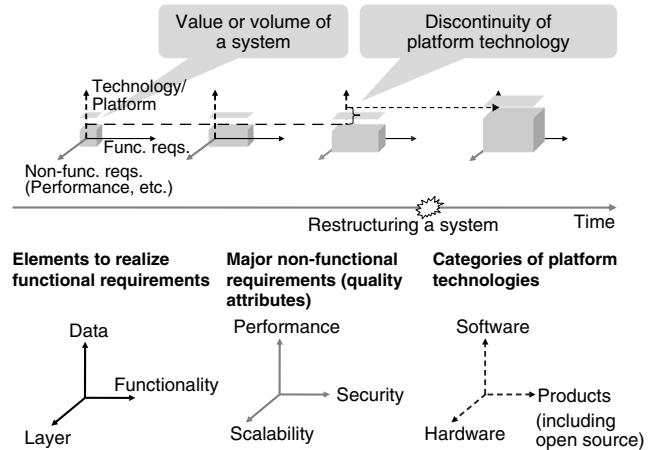
- Variable platform
- Ever-changing requirements
- Weak architectural constraints
- Many codes by many developers
- Long history of made-to-order developments

These characteristics hinder the reuse of software in the long run. We cannot concentrate on application development on such variable platforms and platform-dependent assets are not reusable. Furthermore, we usually not only develop systems but also enhance them throughout their life cycle, and functional and non-functional requirements are never fixed. Figure 1 shows the major concerns of our system development and enhancement, each of which independently evolves with time.

Compared with embedded systems and hard real-time systems, enterprise systems have fewer constraints in the decomposition of a system into modules because hardware is abstracted by middle layers and relatively ample hardware resources are available. Such flexibility makes it difficult to define components in suitable granularity and find criteria for selecting features.

Enterprise software sometimes exceeds millions of lines, which are produced by hundreds of developers working in parallel under weak architectural constraints. Therefore, not only the complexity of the software but also the complexity of the organization is an issue to be considered.

**Figure 1. Capturing Concerns with Time**



Besides, neither data structure nor functionality is identical in our retail industry domains due to few or weak regulations compared with those in the financial industry. Therefore, we usually mine requirements from our clients and develop a system based on those client-specific requirements. This activity has lasted about thirty years so that the made-to-order style of development took root as part of our organizational culture.

Enterprise resource planning (ERP) package software could be a solution to the problems outlined above, but not a primal choice for companies with strong discipline in their business operations, which are processed by information systems specific for them. If we were to apply ERP packages for these companies, we would eventually have to develop countless glue programs to interact with other existing systems and customize the black box (ERP) software with a great deal of effort. Consequently, these companies would not enjoy the benefits of ERP in terms of cost and speed.

# 2 History of Our Challenges

It takes a long time to tackle the characteristics described in the previous section. In 2001, the

Distribution Systems Division of NRI began to develop semi-made-to-order package software following SPL

practices. This package software was meant to be used in the development of systems for retail chains. The ambitious objective of this challenge was to bring a flexible system solution to our client companies using a short lead time and with cost effectiveness.

## 1 | Developing Semi-Made-to-Order Package Software (2001 – 2002)

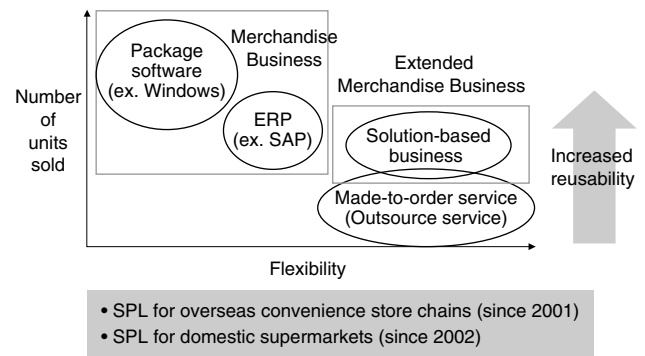
NRI's challenges were inspired by the risks of sticking to our traditional hand-made style of system development, which is sustainable only if our clients are able to pay the costs and accept the lead time required from order to delivery.

In addition, neither data structure nor functionality is identical in our retail industry domains, due to few or weak regulations compared with those in the financial industry. Therefore, creating single-solution software for multiple clients is not feasible in our business.

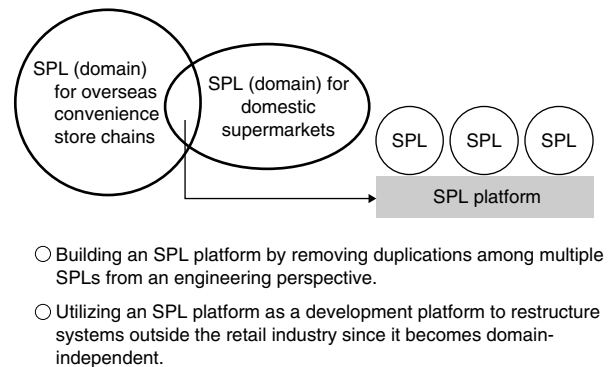
In 2001, our challenges started with the development of semi-made-to-order package software for convenience store chains, since we had to restructure the mission critical systems of our overseas clients at the same time. The following year, we developed the same type of custom ready-made software for domestic supermarket companies including anticipated clients.

Figure 2 is a mapping of the software merchandise business and the segment that we pursued for semi-made-to-order package software.

**Figure 2. Mapping of Software Merchandise Business**



**Figure 3. Building an SPL Platform with an Engineering Perspective**



## 2 | Establishing an SPL Platform and Verifying it Outside of Our Domains (2003 – 2004)

By engaging in the two semi-made-to-order software development projects described above, I realized that large parts of the source code were similar and only slightly different even if the two projects were targeting different domains.

To maximize the overall maintainability of made-to-order software, which were to be vital, or core, assets for our objectives, I began to mine the overlapping

parts and abstracted them to minimize the amount of software development required in each project.

Not only pursuing variability management in a single domain, but also managing variability across domains is essential for executing SPL practices in large scale over a long period of time. We mined the SPL platform that consists of reusable assets. This accelerated software development and eased variability management in the application layer.

The SPL platform consists of design patterns, methodologies specialized for the platform and reusable software that is based on dozens of open source software (OSS) products. The platform itself is independent of application domains so that we validated the independence and effectiveness of the platform by restructuring systems in the wholesale industry.

Figure 3 shows the relationship among multiple semi-made-to-order software packages and the SPL platform.

### 3 Restructuring of Mission Critical Enterprise Systems on an SPL Platform (2005 – 2006)

Around 2005, Seven & i Holdings Co., the largest conglomerate in the Japanese retail industry, needed to restructure the mission critical systems of some of its group companies. NRI had developed and enhanced these systems for each company.

To pursue the agility of its corporate-wide enterprise systems, Seven & i decided to build a standardized group system in the supermarket and restaurant domains instead of restructuring a collection of company-optimized systems. However, a sole standardized group system was not acceptable since a variety of constraints existed among companies operating in the same domain. Therefore, implementation of a variability management mechanism was needed to satisfy mandatory company-specific requirements.

While restructuring the mission critical systems within the two domains, an extended SPL platform specialized for Seven & i Holdings was established. The extended SPL platform is domain- and company-independent but has common features within the corporation, such as security, so that it can be reused in the following system developments throughout the corporation. Figure 4 represents a software stack in applying an SPL approach within the conglomerate.

### 4 Extending SPL Activities Further Inside and Outside Our Domains (beginning in 2007)

We are currently extending our SPL approach beyond our primary domain, the mission critical systems of the retail industry.

For instance, workflow-oriented applications have been developed based on COTS products by accepting some limitations regarding client requirements. Thanks to OSS products, we can quickly learn the best practice in the domain besides executable source codes. By integrating and extending those OSS products, we enable variability management in

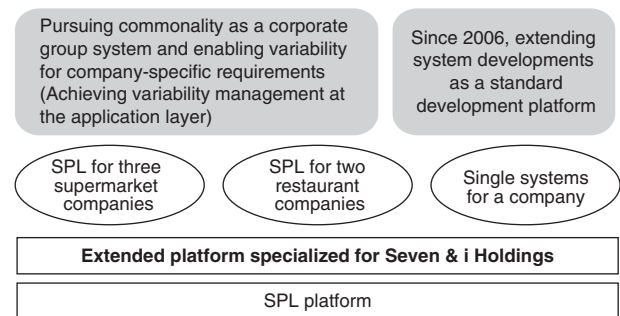
workflow/BPM applications with SPL platform extension.

Not only pursuing other SPL target domains, we are also seeking to improve the speed of application development. Domain-specific language (DSL) is an example of boosting application development under some special conditions. We developed DSL based on an SPL platform and educated our new engineers by focusing on design issues with minimum implementation concerns.

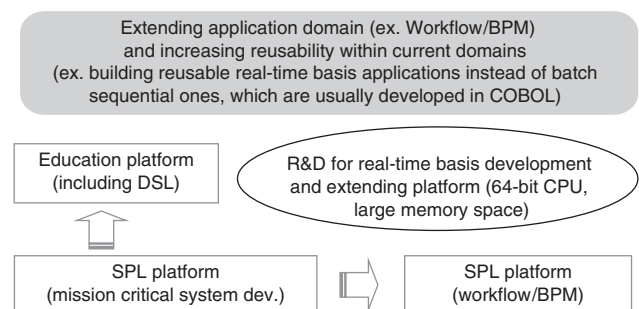
While object-oriented language has prevailed, many business applications continue to be implemented in batch sequential styles using legacy languages such as PL/I and COBOL. These applications are usually hard to maintain with limited reusable mechanisms. Once we have a solid solution for achieving the same logic on a real-time or online basis in solving performance and reliability issues, the logic of a majority of these applications can be implemented on an SPL platform.

Figure 5 summarizes our latest challenges based on an SPL platform, which was originally developed for mission critical system development.

**Figure 4. Applying an SPL Approach within a Corporation**



**Figure 5. Our Current Challenges Based on an SPL Platform**



# 3 Design Strategies

The previous section describes the history of our challenges in pursuing SPL practices for development of enterprise systems. This section summarizes our design strategies, which have been validated through dozens of applications since 2001.

## 1 Handling Three Different Levels of Variability Management at One Time

To practically apply an SPL approach in our business, we must be able to increase the productivity and reusability of an application. Therefore, we must prepare and optimize the architecture for application development by removing all concerns except for that of implementing specific functional requirements.

Based on our experience, our current understanding of an enterprise SPL approach is as follows.

*An enterprise SPL is based on an application-centric architecture, which manages platform variability and non-functional requirements, and achieves variability management for functional requirements in an application layer.*

The characteristic of an enterprise SPL approach is an architecture that enables three levels of variability management at one time and maximizes the overall efficiency of application development. The three levels of variability management are listed and explained in the following sections.

- 1) Variability management of platforms
- 2) Variability management of non-functional requirements
- 3) Variability management of functional requirements

### (1) Variability management of platforms

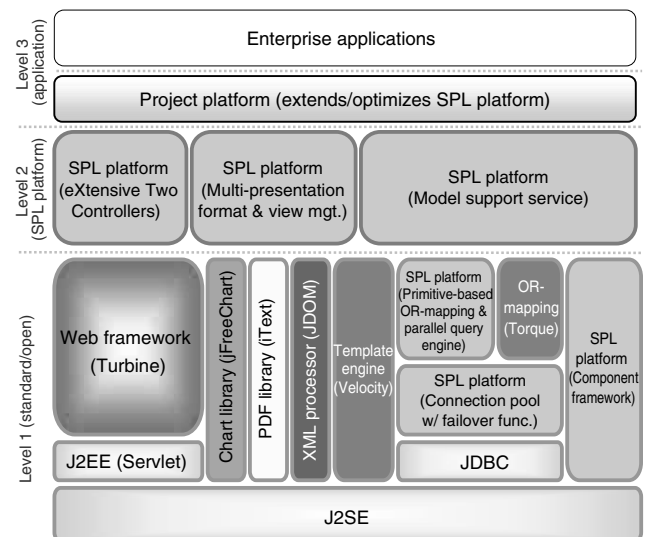
Because platforms of an enterprise system are easily changed, building an application on top of a fixed

platform seems to have no feasibility. To avoid the impact of a platform change to our applications, we must be careful not to build core assets that are directly coupled with platform-specific features.

In discussing the term *platform*, we used to have in mind that a platform constitutes a set of hardware, O/S and some middleware such as relational database (RDBMS) software. However, in the last decade, wide use of OSS products in enterprise systems has significantly expanded the notion of *platform*. For example, these days, a variety of web application frameworks is regarded as a part of a platform. OSS products evolve very rapidly, and if we do not have smart ways to adapt our applications to the changes of the OSS products, we must expend a lot of effort to be able to enjoy the benefits of the evolution of OSS products.

As shown in Figure 6, we provide a middle layer, which is named an "SPL Platform," between application and standardized platform products including OSS products. The main objective of this layer is to manage the variability and changeability of the platforms by removing dependencies between applications and platforms.

**Figure 6. Variability Management of Platforms**



Another objective of the middle layer is to extend and customize platform products in order to provide optimized interfaces for application development. While our experience indicates that the latest OSS products are not always the best choices, we should pay careful attention to the architectural conformance of each OSS product to an overall architecture.

In addition, we adopted thin clients so that we could manage the platform variation of client PCs with less effort. We also developed RDBMS-independent applications by using this SPL platform, which manages the variability of RDBMS.

## (2) Variability management of non-functional requirements

The more that network access is widely available, the more are enterprise applications designed for the online- and data-centered architectural style. In contrast, mainframe-based enterprise legacy systems were designed for the offline and batch sequence architectural style.

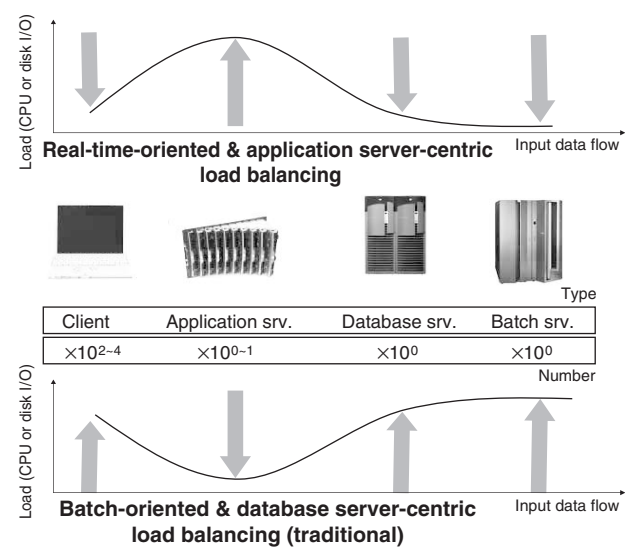
Due to this paradigm shift in architectural styles, enhancements of enterprise systems have become difficult through their life cycles with insufficient management mechanisms for non-functional requirements. Moreover, the non-functional requirements related to data and transaction volume vary from client to client and the requirements usually change along with each system's life cycle.

To maximize the reusability of applications that realize functional requirements, non-functional requirements must be managed without having an effect. Therefore, managing the variability of non-functional requirements without depending on a specific platform and interfering with applications is the inevitable criterion for this approach.

We need system-level architectural strategies to realize our objective, which result in "an inversion of load balancing." Figure 7 compares traditional batch-oriented (below) and real-time-oriented (above) load balancing approaches.

In our load balancing approach for scalability, all transaction loads are accumulated at application (AP) servers as much as possible and distributed among

**Figure 7. Variability Management of Scalability**



them to avoid performance bottlenecks at RDBMS, which are usually difficult to be distributed due to data integrity. To satisfy the response time requirement, we usually de-normalize tables at RDBMS for real-time transactions, which tends to make our system inflexible. Meanwhile, our SPL approach is different in that data are not duplicated to other tables for response. To enable this approach, we built specialized services within our SPL platform, some of which have similar functionalities that a database server would provide.

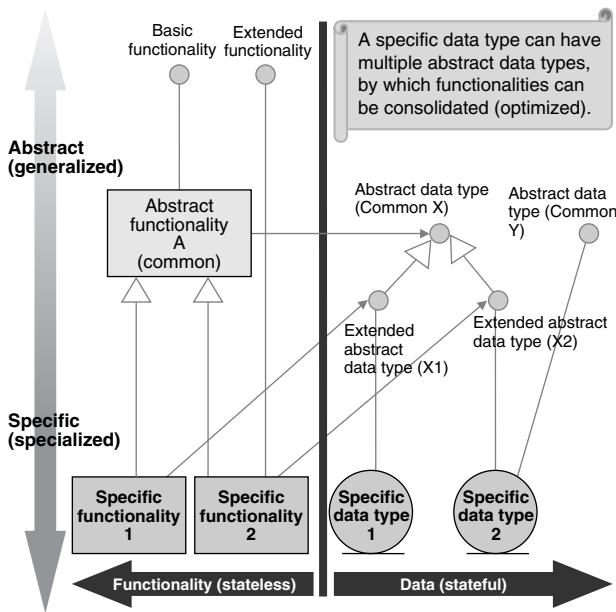
## (3) Variability management of functional requirements

By separating the concerns regarding platform and non-functional requirements from the concerns over application reuse, the basis for building context-free and platform-independent assets is finally ready.

Since the required functionalities are different for every client and evolve through a system's life cycle, the data structures used in the system will also be required to change along with functionality variation and evolution.

However, the relationship between functionality and data structure is not necessarily one to one. Therefore, the independent evolution of software along both the functionality and data dimensions must be supported in a noninvasive manner. To achieve such independent evolution, applications must be modularized in

**Figure 8. Variability Management of Functionality and Data**



appropriate granularity and loosely coupled by the following techniques, information hiding and the abstract data types (ADTs). These key techniques are fundamental and must be sustained for a long period and have a high degree of potential reuse according to the functionality and data dimensions of the separation of concerns.

Figure 8 presents these techniques in separating functionalities from data types and sharing both functional and data type commonalities by abstraction.

From the point of view of development, both functionality and data modules are usually separated further along additional dimensions such as layer and reuse. From the point of view of deployment, these modules, a set of specific types, are instantiated by an inversion of control (IoC) container, a basic mechanism of pluggable architecture.

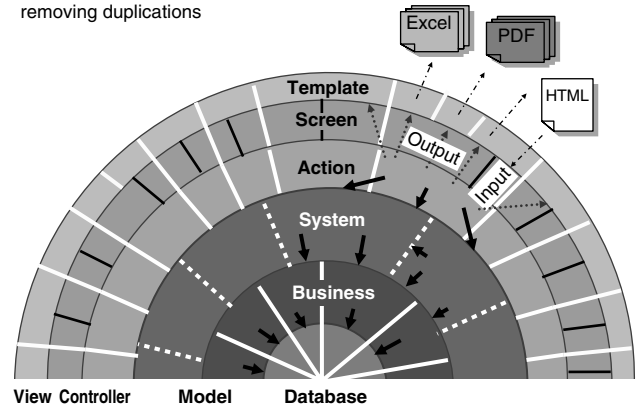
A more detailed explanation of the architecture that enables variability management for functional requirements will be covered in the following section on design strategy, MDSOC.

## 2 Applying the MDSOC Approach

“Handling three different levels of variability management at one time” is the first step and overall design

**Figure 9. 2MV2C Layered Architecture**

- Data-centric layered architecture style
- Hiding information & removing duplications



strategy for enabling an SPL approach in enterprise system development. As the second step, we explain the method of decomposing application modules.

With a procedural language, such as COBOL, we usually decompose modules based on functionality. Conversely, a data-oriented approach or object-oriented principle methodology is usually based on data. Either approach is not always appropriate since we need both points of view to decompose a system simultaneously. Otherwise, we shall be faced with the tyranny of dominant decomposition.

Multi-dimensional separation of concerns (MDSOC), simultaneous decomposition, is an essential design strategy for the development of long-term sustainable and reusable assets. Besides functionality and data dimensions, three more dimensions—layer, granularity and reuse—are added to decompose software into modules.

To maximize reusability and minimize redundancy, a Model-view controller (MVC) pattern is extended to 2MV2C (2-models, 1-view, 2-controllers), a closely layered structure, in which all functionalities are packaged into the five layers and so are data structures.

Figure 9 illustrates the dependencies between the layers. The idea behind this layered architecture is that the more stable modules reside closer to the center. Therefore, a lower layer must not have concerns with higher layers. By following this module mapping principle, we consider views as variations of the model, which is the root of this “onion” architecture.

Figure 10. MDSOC from Functionality Point of View

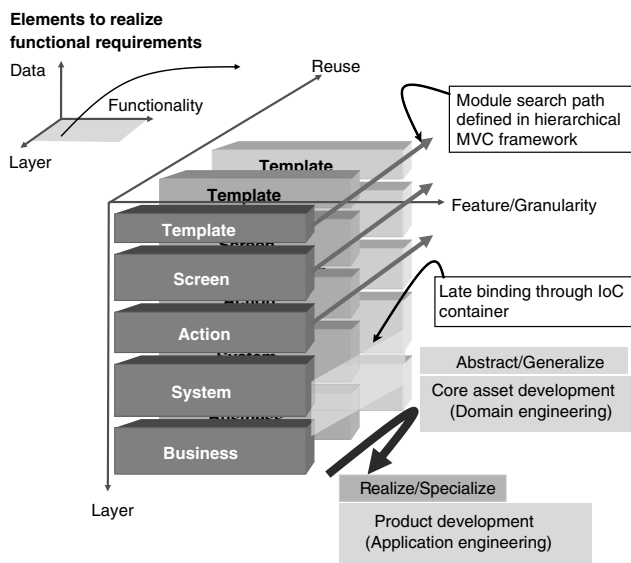


Figure 10 presents the five-dimensional decomposition space from the functionality point of view. A hierarchical MVC approach is applied to the reuse dimension, which is the key dimension in terms of variability management of the functional requirements.

### 3 Resolving a Variety of Impedance Mismatches Throughout a System

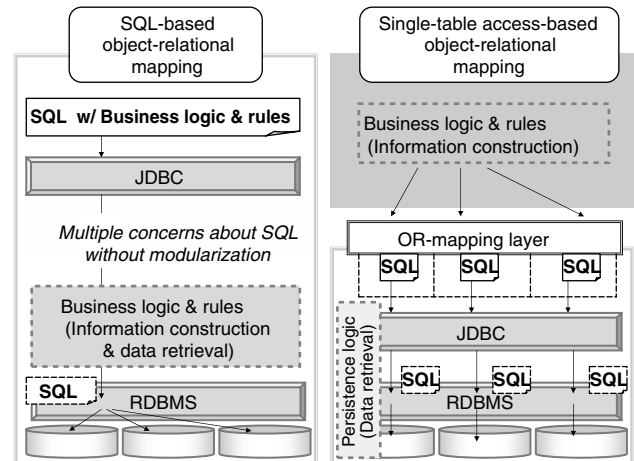
Developing a large enterprise system is just like a war we are fighting with ever-increasing complexity. Due to the large number of applications, which are developed by many developers simultaneously, we need reliable disciplines to put the complexity under control.

Besides three levels of variability management and the MDSOC approach, we must establish precise disciplines for issues of implementation based on layered dependency and scope management.

If layered dependency and scope management are not handled appropriately, the complexity of the application source codes is usually increased. The source code of a mapping object and relational database is a prominent example in considering complexity.

We call impedance mismatch such a mapping problem, in which two different worlds, object and

Figure 11. OR-Mapping Problem and Complexity Control



relational, meet in the middle of an application. The optimal way for handling data is different for each. Therefore, we must alleviate the conflict with fixed disciplines.

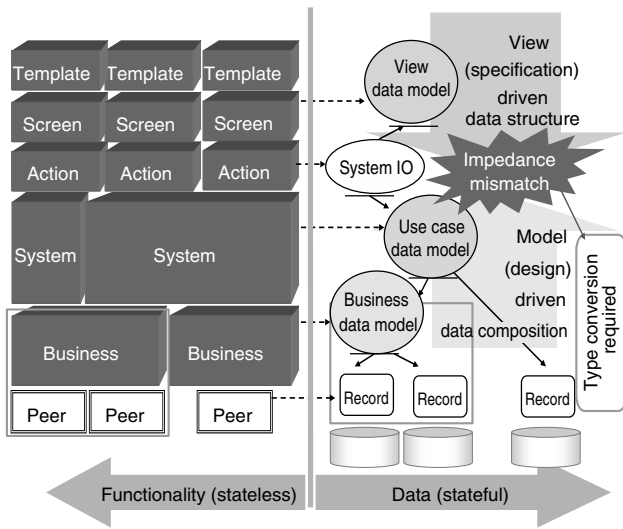
To avoid mixing several concerns in a single module, the fixed discipline restricts the developer to use only single-table accesses, whose source codes are automatically generated. In this approach, a module's scope is reduced to single-table access, and data retrieval and information construction are separated into two different layers.

Figure 11 compares two different approaches to data query. With the reduced scope of each module and divided layers, the complexity of data mapping is converted into a number of simple modules.

Impedance mismatch is not necessarily restricted to the issue of data mapping between objects and relational databases. Another impedance mismatch exists between view and model. View is primarily based on usability so that a data type optimized for view is not always appropriate for model. This discrepancy must be resolved by type conversion while data is being translated between view and model.

Figure 12 illustrates that logic or a functional procedure is decomposed into modules along with layer and scope dimensions while each decomposed logic space requires an optimized data structure. In a model layer, the record data type is the smallest granularity; other data types consist of hierarchical composition of the record type. However, a view data type cannot

Figure 12. Type Mismatch at Layer and Scope Dimensions

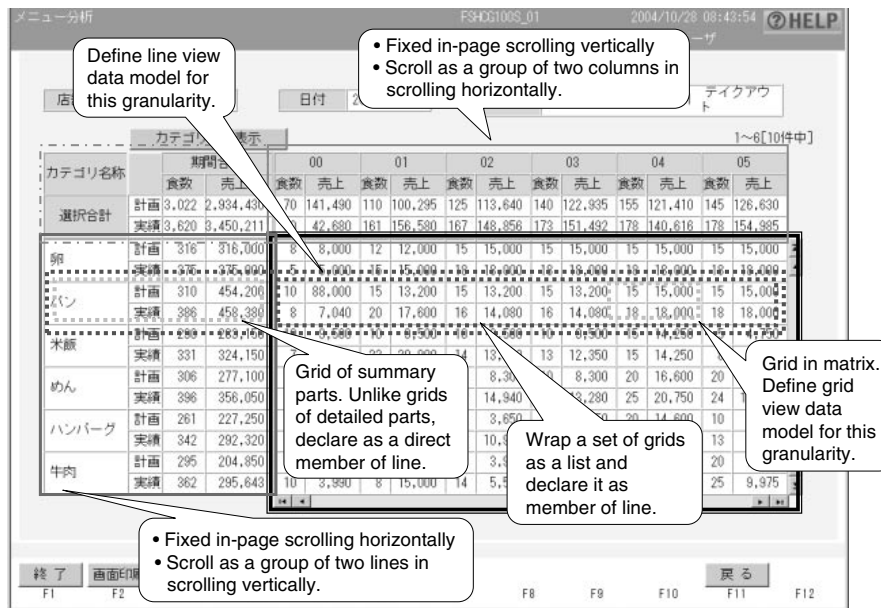


be derived from the composed data type but from the point of view of usability or human-computer interaction.

Type conversion between view and model is mandatory when the difference between matrix and list data structure is considered. Figure 13 is an example of matrix data representation in which GUI-flavored types such as grid and line are required to control the view state effectively. Besides, the matrix is never identical with the data structure in the model since a database manages data as a list.

Finally, a model must not be influenced by any specific view representation. Therefore, this type conversion is accomplished by a corresponding screen module, which resolves the impedance mismatch between its view and the model it represents.

Figure 13. View Data Model (Grid, Line, ...)



# 4 Evaluation and Lessons Learned

Through our challenges over the last six years, we value the following points for successfully applying the SPL approach in enterprise system development.

- 1) Executive management's commitment and feedback from the field
- 2) Balance of speed between training and development

### 3) Role of application architect

#### (1) Executive management's commitment and feedback from the field

For commercially successful SPL implementation in enterprise system development, long-term investment is necessary since we may need to change the corporate culture itself. If a long history of made-to-order system development exists in a corporation, a top-down approach is also required.

Furthermore, reusable software assets are not sufficient to generate returns from SPL. We also need people who can utilize and refine the software assets. Training requires time and money, so executive management's commitment is required to understand and support these activities.

Most companies spend a great deal of money for R&D but the product of the clean room is very rarely applied in the actual development field. SPL is also not supported from such a clean room environment. Building a feedback loop is an essential activity for gradually establishing core assets by verifying their effectiveness through actual operations.

The most important role of executives is continuously and devotedly providing real projects for those who execute SPL practices for a long time.

#### (2) Balance of speed between training and development

There is a big difference between made-to-order system development and the SPL approach. The key difference is abstraction. For an organization, which has developed systems based on specific and distinct requirements, reeducation of employees regarding modeling and abstraction is mandatory. Education

alone is not enough; modeling and abstraction must take root in the organization's culture.

In addition, pure waterfall-type system development is not feasible while core assets in the application layer are not yet matured. While those assets are maturing, intensive communication is required among project members throughout all development phases.

Once short-term development cost reduction is given increased priority, traceability or the relationship between modeling and implementation becomes very difficult to maintain, which eludes the reusability of core assets. Since detailed design decisions are usually completed during the implementation phase, it is strategically important to craft software for reusability.

#### (3) The role of application architect

The key to success in the enterprise SPL approach is variability management at the application layer. To achieve this goal, we need an application architect who has the ability to structurally design an application layer with an SPL platform based on domain knowledge.

The more an SPL platform is evolved and matured, the more important the application layer becomes. The important role in terms of generating value shifts toward the application layer from platforms or middleware itself, in which an application architect is required to define the appropriate granularity of components for effective development, evaluation, enhancement and reuse.

In reality, however, perfect design cannot be accomplished from the beginning. Continuous refinement or re-factoring is the most practical approach while a grand design must be fixed during an early phase.

## 5 Conclusions

Our challenge still remains in the early phases of progress toward our objective. On our SPL platform, we must establish more specific connectivity between requirement engineering and core asset development

in the application layer. Currently, use case modeling and data modeling are primarily applied for non-user interface issues. However, these approaches are not powerful enough to capture variability.

In the meantime, as SPL platforms evolve and specialized design patterns mature, implementation tasks are further reduced and implementation moves closer to actual design. In this context, those who know requirement specifications are the right persons for implementing executable source code. During this present year, we will see the potential of this type of development.

Our objective is not to maximize the steps to be reused but to deliver more value to our clients in a timely manner by maximizing cost effectiveness in our

development efforts. Educating and managing many developers is one approach but developing large systems with a limited number of specialists could also be an option.

We consider system development on this ever-evolving SPL platform just like art. Art is fun. In considering the motivation and enthusiasm of the employees in our division, in-house development may be an ultimate goal as long as our in-house development team develops a system faster, cheaper and of higher quality than any other approach does.