14TH INTERNATIONAL DEPENDENCY AND STRUCTURE MODELING CONFERENCE, DSM'12
KYOTO, JAPAN, SEPTEMBER 13 – 14, 2012

ARCHITECTURE MIGRATION USING DSM IN A LARGE-SCALE SOFTWARE PROJECT

Takashi Maki
Corporate Technology Development Group, Ricoh Co., Ltd., Japan

ABSTRACT
In product line development (PLD), maintaining architecture against the rapid changes in market requirements and the technical environment is important. When maintaining architecture, architecture migration is often necessary to keep the architecture up to date. An issue in architecture migration is the continuity of the products. Because architecture is the basis of the development of core assets, architecture migration can result in core asset redesign or reimplementation, which can have a big impact on existing products. In large-scale PLD, this problem becomes very serious. Furthermore, our project required architecture migration along with real product development. This short paper outlines our approach towards architecture migration of effectively using DSM for a large-scale software project.

Keywords: Architecture migration, product line development

1 INTRODUCTION
In product line development (PLD), it is essential to determine the appropriate product line architecture (PLA) and maintain conformance of the implementation to the architecture [1, 2]. However, in projects with a long life cycle, changes in the architecture cannot be avoided owing to changes in requirements and technologies, and discrepancies between the implementation and architecture can emerge. PLA migration is necessary in order to adapt to changes in the requirements over long life cycles. As we mentioned in [8], continuous and planned refactoring is thus essential to lengthen the life of a PLA. In [8], we discussed the issue of planning the strategy for PLA migration along with product development. Determining the PLA migration strategy is a necessary and important factor when determining the benefits of the migration.

In large software projects, we believe that drastic changes should not be made without considering continuity from the former architecture. In general, architecture migration can result in the redesign or reimplementation of core assets, which can have a big impact on existing products. In large-scale PLD, this problem can be very serious. From this viewpoint, we planned architecture migration using the source code of products from the former architecture to the new architecture. The new architecture is determined by referring to the former architecture in consideration of product continuity. Actual activity of the migration on the source code includes refactoring, such as splitting modules or moving modules, in order to achieve conformance between the implementation and the new architecture. In this situation, detecting and managing the discrepancy between the implementation and new architecture is an effective method. This paper reports an example of using DSM to maintain software.

The rest of the paper is organized as follows. In Section 2, we describe the motivation for architecture migration and for using DSM in our architecture migration. In Section 3, we explain our experiences with architecture migration using DSM. In Section 4, we discuss the results. We conclude this paper in Section 5.

2 MOTIVATIONS
The target source code that we applied DSM to belongs to controlling software of a digital still camera. Because of the market needs, a new product is released almost every six months. Sometimes, more than two products are developed concurrently. Although each product is developed based on a common architecture, opportunistic reuse and unplanned extensions during product development have

led to several problems in clone-and-own development, which has made concurrent development of multiple products on a single platform difficult. Although we have developed products using the common architecture, it is not up-to-date in terms of adapting to market requirements and the technical environment. This was our motivation for considering architecture migration. In the target product domain, we thought partial modification of the architecture would be more effective than developing new architecture from scratch for the following reasons:

- Because the products are released at short intervals, there is not enough time to reconstruct the overall software.
- There is strong product continuity before and after architecture migration.

Static software architecture is often expressed by defining the function group and the dependencies between them. Once we define a new architecture, it is quite normal that there are discrepancies between the new architecture and the former implementation. What is important is how we detect these discrepancies and how we reduce them. This is the procedure called architecture migration. One strategy for architecture migration is to organize a special task force for migration. However, we took a step-by-step architecture migration strategy along with ordinal product development because of the restriction of product development. In this situation, we encountered the necessity of the following tools:

- Automated method to detect discrepancy of dependencies for implementation
- Simple and visual expression of these discrepancies

These were our motivations for applying DSM in our architecture migration.

3 ARCHITECTURE MIGRATION

3.1 Defining the new reference architecture

We redefined the architecture by referring to the previous architecture with a similar layer structure. Figure 1 shows an overview of the redefined architecture. Although the architecture before migration did not have explicit layers, the role and relationship of the layers were defined in the new reference architecture. In Figure 1, each box with text represents a top-level subsystem. Some of the adjoined layers are merged for simplification, and detailed subsystems and components are filtered out. Layers were arranged roughly according to the abstraction level. Lower layers mainly handled hardware controls, while upper layers mainly handled the user interface. The general idea was derived from the former architecture, but, in order to clarify the role of each layer, some layers were divided into two or more layers. The relationship between layers expresses the dependency between layers. Because strict layering was not enforced, the access directions from top to bottom and left to right were defined as allowed access; these are shown as solid arrows in the figure. The opposite directions, shown as broken arrows, were defined as disallowed access.

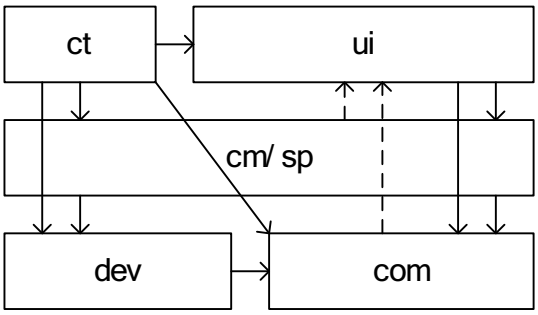


Figure 1. New reference architecture

3.2 Migration using DSM

To analyze access between top-level subsystems, we determined the directory structure of the source code in our coding guideline to correspond to the top-level subsystem in the reference architecture and the first-level directory in the implementation. Figure 2 shows an image of the directory structure of the implementation. In Figure 2, the second- and lower-level directories are omitted.

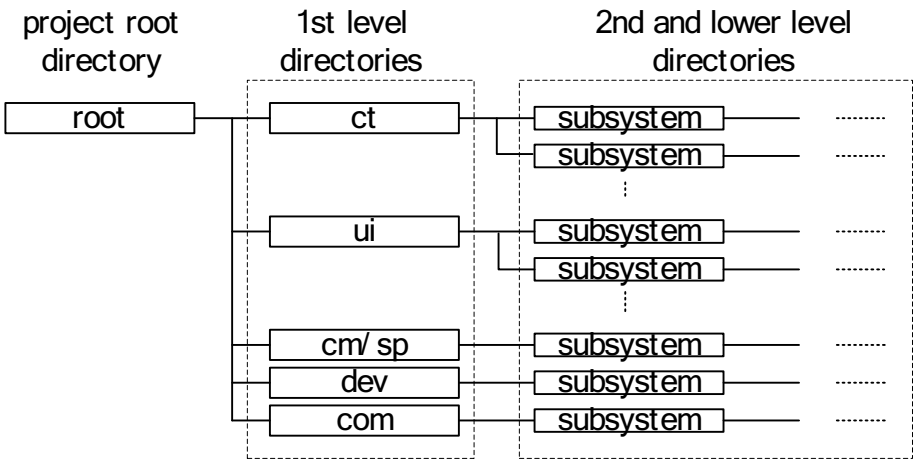


Figure 2. Directory structure of the implementation

By maintaining the correspondence between the directory structure and top-level subsystem, we found it easier to analyze the caller–callee relationship between top-level subsystems using DSM tools such as Lattix [9]. Figure 3 shows an image of the DSM for the implementation at the start point of the architecture migration. The numbers in each matrix cell are dummies to preserve confidentiality. In this DSM, the order of directory entries was arranged to reflect the determined access direction of the reference architecture. In this DSM, the lower-left triangle represents the allowed accesses, and the top-right triangle represents the disallowed accesses. Access is comprised of calling functions, including headers and referencing variables. In our architecture migration, we controlled the number of top-right cells and their content to reduce the disallowed accesses.

$root			ct	ui	cm/sp	Dev	com
			2	3	4	5	6
root	ct	1		13	49	6	
	ui	2	13			2	
	cm/sp	3	62	31		31	7
	dev	4	81	1	53		5
	com	5	31	3	2		

Figure 3. Example of DSM

3.3 Results of the migration

After determining the reference architecture shown in Figure 2, we listed the instances of inappropriate access by using the DSM tool [5] and then began correcting those instances. This correction involved moving software parts between layers, etc. For performing the correction, we referred to known techniques such as [6][7]. We called all such instances of inappropriate access as "access violations." For the metric, we measured the "violation ratio" (VR) of the code base regularly. VR is calculated as follows:

VR = (number of access violations)/(number of total accesses) × 100 [%]

Here, the number of total accesses is the sum of the number of access violations and allowed accesses. Figure 4 shows the transition of VR along with the product development. P4 is the product before the start of the architecture migration. P5–P8 represent the products following P4. The vertical axis was normalized to 1 for the value of VR at P4. P8 had an almost zero VR value and was actually close to zero. To observe the transition, we used some available tools such as those introduced in [5][9]. In the project introduced in this paper, checking was conducted using the reverse engineering tool introduced in [5]. The organization continued product development over three years by using the platform created by these steps. More than 10 products were derived from this product line infrastructure, and the number of access violations reduced through regular inspection and correction.

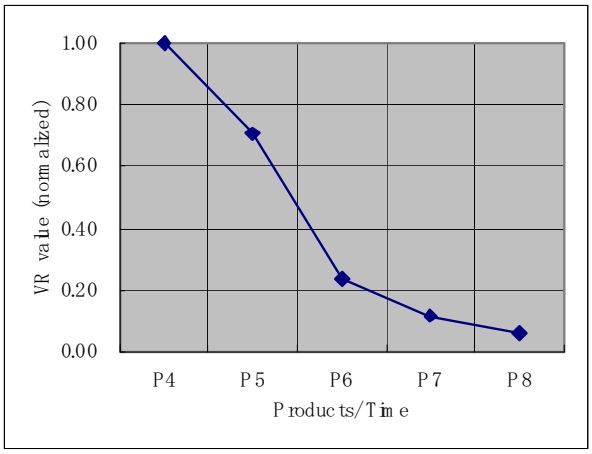


Figure 4. Transition of violation ratio

4 DISCUSSION

4.1 Effectiveness of the approach

We found that this approach was effective for a practical project in terms of maintaining the continuity of the product before and after architecture migration. The VR metrics clearly indicated the progress of correcting the implementation. To repeat the listing of access violations at short intervals, an automating tool was needed to extract accesses in the project and visualize them. For this purpose, a combination of tools [5][9] was used to occupy a key role in our project.

4.2 Other findings

The other findings from our experience were as follows:

- Limited adoption of strict layering. Strict layering brings many benefits that can be obtained from the layered structure if feasible. However, there are tradeoffs in terms of system performance and memory consumption, which are both important issues in embedded systems. For such a layered structure, upper layers tend to be exposed to change requirements because the upper layers handle user function requirements; in contrast, lower layers mainly handle hardware, which is relatively stable over products. Thus, the dependency from the lower layers to upper layers is problematic because there is a possibility of destabilizing the lower layers. On the other hand, the jump in dependency from the upper layers to lower layers is less critical compared to the reverse dependencies.
- Ease of inspection. In order to standardize implementation in a large project, a simple and easy inspection method is necessary. Thus, simple metrics based only on the source code configuration, such as VR, are an easy inspection method.
- Educational effect. DSM shows us the outlook of internal access in the project in a simple and visual way. As a result, the necessity of correcting access violations are correctly communicated to all project members, including newly joined members who are not familiar with the project situation.

5 CONCLUSION

In a large and long life cycle product line, architecture migration becomes important. In this short paper, we propose architecture migration techniques for such product lines and report the results of applying the techniques to a practical project. We found that DSM can be effectively used for such architecture migration in a large-scale software project.

REFERENCES

[1] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*, Addison-Wesley, 2001.

[2] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice* (2nd Edition), Addison-Wesley, 2001.

[3] J. Knodel, D. Muthig, et al., Architecture Compliance Checking: Experiences from Successful Technology Transfer to Industry. *European Conference on Software Maintenance and Reengineering*, 2008, pp. 43-52.

[4] D. Muthig, Bridging the Software Architecture Gap. *IEEE Computer*, June 2008, 98-101.

[5] N. Sangal, Lightweight Dependence Models for Product Lines. *10th International Software Product Line Conference*, 2006, p. 228.

[6] S. Roock and M. Lippert, *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*. John Wiley & Sons, 2006.

[7] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[8] T. Maki and T. Kishi, Problem Factor Portfolio Analysis for Product Line Architecture Refactoring. *17th Asia Pacific Software Engineering Conference*, 2010.

[9] Understand, http://www.scitools.com/.

Contact: Takashi Maki, Ricoh Co., Corporate Technology Development Group, 16-1, Shinei-cho, Tsuzuki-ku, Yokohama 224-0035, Japan, e-mail: takashi.maki@nts.ricoh.co.jp

Architecture Migration Using DSM in a Large-Scale Software Project

Takashi Maki

Corporate Technology Development Group, Ricoh Co., Ltd., Japan

Technische Universität München iSiD IT Solution Innovator

BY MANAGING COMPLEXITY GAIN COMPETITIVE ADVANTAGE

RICOH imagine. change.

Index

RICOH
imagine. change.

Motivations

- Target product domain
  - Control software on compact digital still cameras
  - Products are released 5-8 models are released per year.
  - All products are developed using the common architecture
  - Strong needs to migrate architecture to adapt to.
    - Market requirement
    - Technical environment

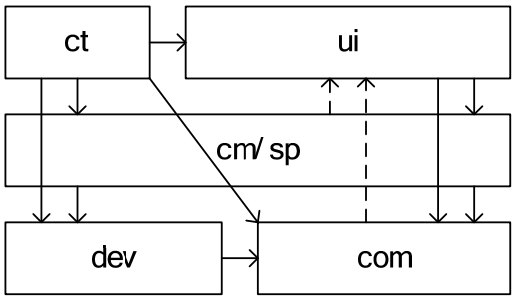
RICOH
imagine. change.

Motivations

- Conditions
  - Because the products are released at short intervals, there is not enough time to reconstruct the overall software.
  - There is strong product continuity before and after architecture migration.
- Requirement for tool
  - Automated method to detect discrepancy of dependencies for implementation
  - Simple and visual expression of these discrepancies

Architecture migration

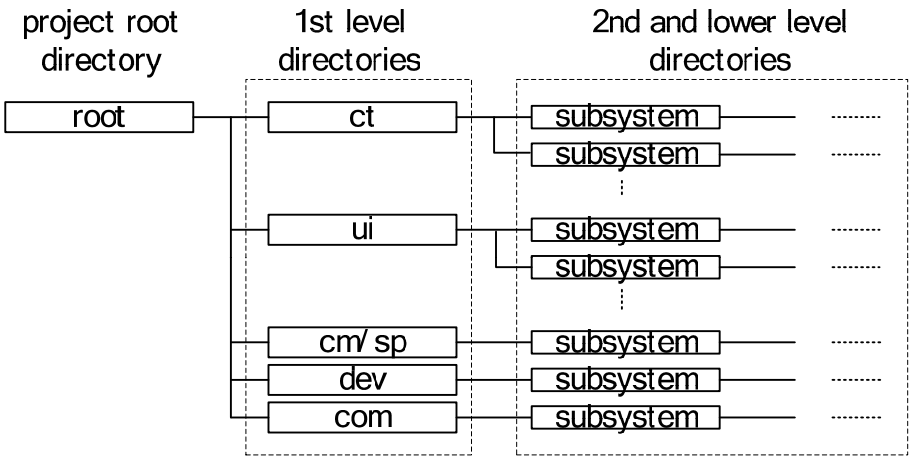
- Defining a new architecture
  - Defined by referring to previous architecture
  - Layers are introduced
  - Strict layering were not enforced



New Reference Architecture

BY MANAGING COMPLEXITY GAIN COMPETITIVE ADVANTAGE
RICOH imagine. change.

Architecture migration



Directory structure of the implementation

Migration using DSM

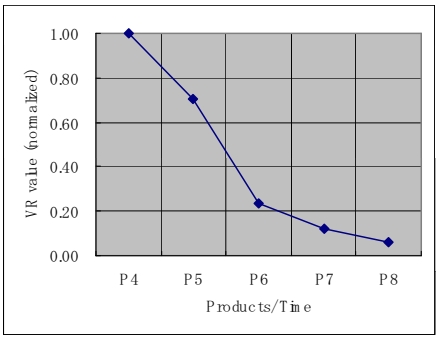
- Access counts …
  - Including header files
  - Function calls
  - Referencing variables
- Implemented file directory structure corresponds to the static structure of the architecture

$root			ct	ui	cm/sp	Dev	com
			2	3	4	5	6
root	ct	1		13	49	6	
	ui	2	13			2	
	cm/sp	3	62	31		31	7
	dev	4	81	1	53		5
	com	5	31	3	2		

Example of DSM

Results of the migration

- Metrics “VR” (violation ratio)
  VR = (number of access violations)/(number of total accesses) x 100 [%]
- Correcting process
  - Measured “VR” in each product, then fixed at the next product



Transition of “VR”

RICOH imagine. change.

Discussion

- Effectiveness of the approach
  - Effective if continuity of products exists
  - To repeat the listing of access violations, tool support was essential (LATTIX)
- Other findings
  - Layering, but not strict
  - Ease of inspection
  - Simple metrics “VR” (violation ratio)
  - Educational effect
- Key for success in improvement on implementation quality
  - Careful determination of the new architecture
    - Consideration of product continuity
  - Persistent check of violation
    - Check at each product

RICOH imagine. change.

Summary

- Adapted DSM to a large-scale software product line
- Succeed in improving software implementation quality