# Studies on Architecture Refactoring for Software Product-Line Development

by

Takashi MAKI

submitted to
Japan Advanced Institute of Science and Technology
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

*Supervisor:* Professor Tomoji Kishi

*School of Information Science*
*Japan Advanced Institute of Science and Technology*

December, 2013

# Abstract

In recent years, it has become increasingly important to maintain architecture in product-line development (PLD), mainly because of the rapid changes in market requirements and technical environments. In PLD, architecture maintenance is a more complicated and difficult process compared to conventional software development because architecture is key to achieve large-scale reuse in developing a product family. In architecture maintenance, we have to consider both the reference and implemented architectures. Here, reference architecture is a design intention that constraints the implementation, and implemented architecture is an abstract expression of the existing implementation.

Architecture maintenance includes both keeping the conformance of implemented software architecture with the reference architecture and changing the reference architecture to meet new requirements. These architecture changes are modifications of software structure without changing the major feature of the product family. Thus, we call such modifications architecture refactoring.

In PLD, requirements for reference architecture can change during the development of the product family because the development period lasts longer than that in non-PLD. Moreover, the implemented architecture can deteriorate over the development of multiple products. Therefore, we can organize architecture refactoring more efficiently by separately considering the implemented and reference architectures refactorings. In this study, we propose a decision taking method for architecture refactoring that considers both the implemented and reference architectures separately.

The main characteristic of this method is utilizing the portfolio analysis of the problem factor to organize the architecture maintenance strategy. Furthermore, we verified the effectiveness of the proposed method by applying actual project data to the proposed method retroactively.

# Acknowledgements

The author would like to thank Professor Tomoji Kishi for his continuous support and suggestions to the research. The author would also like to thank Professor Koichiro Ochimizu, Professor Mizuhito Ogawa, and Associate Professor Toshiaki Aoki of Japan Advanced Institute of Science and Technology, and Professor Yoshiaki Fukazawa of Waseda University for their valuable reviews, advices, and suggestions. Associate Professor Masato Suzuki gave us helpful comments regarding the analysis of source codes. Associate Professor Defago Xavier provided helpful comments in planning the outline of the research. This research is motivated by the architecture refactoring experiences in consumer product projects. We appreciate every member who participated in the projects.

Finally, I would like to give my special thanks to my family members, especially my wife, whose encouragement and cooperation enabled me to complete this thesis. I would not have been able to finish this thesis without my family's support.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

For the efficient development of a series of products that have similar characteristics, the Software product line (SPL) [9] technique is widely used in various areas, especially in embedded software area. Composing SPL assets and product development using SPL assets is called Product-line development (PLD).

In general, software architecture is important in software development because architecture determines various restrictions on development. In PLD, software architecture is more important for achieving large-scale reuse in developing product families. In case of non-PLD, architecture quality mainly affects product specifications. In case of PLD, architecture quality also affects software reusability. Therefore, many architecture evaluation techniques have been proposed [3] [20] [21].

It is ideal to determine the architecture in advance for the development of all products in the product scope. However, it is often difficult to retain the same architecture during the product scope owing to the rapid changes in the business and technical environments during relatively long periods of development in PLD. Therefore, architecture evolution or refactoring is important for PLD.

In PLD, there are roughly two approaches, reactive and proactive, to compose product-line assets. In the proactive approach [8][25] , the necessity of architecture evolution often arises because of changes in the business and technical environment that occur after using the product-line core asset. Further, in the reactive approach [10], incremental evolution of architecture is necessary because core assets are created by evolving architecture from necessary parts. Therefore, in PLD, architecture evolution is necessary in either approaches.

In this study, we clearly distinguish reference architecture and implemented architecture. Reference architecture represents the design intention of the software structure, and is referred as the design target in implementing software. Implemented architecture is an abstract structure that is realized by source codes of each product. When architecture evolution is considered, we have to pay attention to reference architecture evolution, implemented architecture evolution, and the consistency between both architectures. In this study, we define reference architecture evolution as the change of the design target for adapting new situations, and implemented architecture evolution as the improvement of implementation in order to improve

implementation quality. Because the range of influence and cost differ depending on the kind of architecture, it is important to take decision appropriately when and how to evolve the architecture by considering the situation of product development.

We consider these architecture evolutions as architecture refactoring in the broad sense, and we propose a decision taking method for architecture refactoring by focusing on the difference between reference and implemented architectures.

The rest of this paper is organized as follows: In Chapter 2, we describe the issues obtained through our experience in project operation. In Chapter 3, we summarize the problems that we will solve using our proposed method. In Chapter 4, we propose an architecture refactoring technique. In Chapter 5, we describe the environment in which we examined our technique on the project. In Chapter 6, we present the result obtained using our technique and confirm the past instances using portfolio analysis. In Chapter 7, we show the difference between this study and related works. Finally, we conclude this paper in Chapter 8.

# Chapter 2

# Architecture Refactoring

In this chapter, we describe issues that are background of our research. In section 2.1, we point out problems that are often found in recent embedded software development especially for consumer products. In section 2.1, we summarize general ideas of software architecture that are already known. In section 2.2, we explain reference and implemented architecture that is our important idea in this study. In section 2.3, we describe relationship of PLD and architecture. In section 2.4, we explain difficulty of maintaining architecture. In section 2.5, we describe the usage of utilizing existing artifacts. In section 2.6, we explain architecture refactoring. In section 2.7, we show an example of refactoring opportunity along with the product development.

## 2.1 Software Architecture

It can be seen that software architecture in development means a set of design restriction to keep overall appropriateness of software structure. In other words, software architecture is an abstracted expression of software structure. The term “software architecture” is used to indicate software structure in various abstraction levels. Following are the examples:

- Architecture pattern level

  Architecture pattern means typical pattern of assignment or roles and relationship between elements. As a representative of architecture pattern, layers and broker and so on are introduced in POSA. Also in GOF, several architecture patterns are introduced.

- Functional decomposition and dependency at subsystem level

  At this level, software architecture mainly determines functional decomposition and dependency among subsystems in an architecture pattern.

- Local structure level in implementation

  At this level, software architecture mainly determines local structure inside subsystems.

In this study, we focus on functional decomposition and dependency at subsystem level. Subsystem level means coarse functional unit that is got at conceptual design phase, and not mean local structure of implementation such as class or method.

## 2.2 Reference and Implemented Architecture

As abovementioned, software architecture is an abstracted expression of software structure. In this study, we distinguish the reference architecture, the implemented architecture, and the implementation as illustrated in Figure 1. Definition of these two architectures and the implementation are as followings:

- Reference architecture
  Architecture as a design intention
- Implemented architecture
  Abstracted structure of the implementation
- Implementation
  Substantial artifact such as source code and directory structure

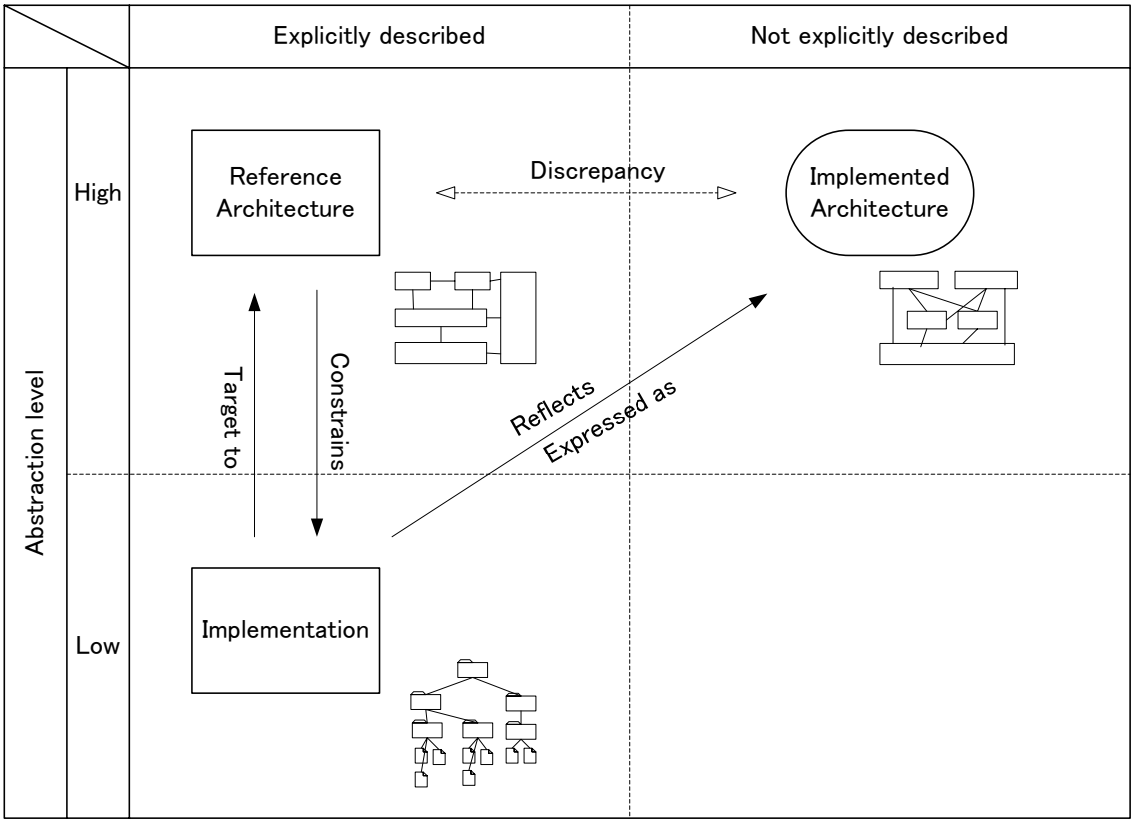


Figure 1 Relationship between General Ideas for Architecture

Generally in software development, we plan a rough structure by considering various trade-offs, at early stage of development. The word “architecture” sometimes indicates the structure defined at this stage. In many cases, source code is implemented by referring the architecture determined at this stage. In this study, we call this architecture as “reference architecture”. Ideally consistency between the reference architecture and the implemented architecture should be kept throughout the development. However the gaps between them often appear. About the reasons of appearing gaps, we explain in section 2.4.3 . In order to investigate the gap, it is useful to understand the structure of the implementation at similar abstraction level of structure. We call this structure of implementation as “implemented architecture”. In other words, implemented architecture means the “architecture of the implementation”. By making distinction between reference and implemented architecture, we can grasp problems on these architectures and problems between them. Details of these architectures are followings:

- Reference Architecture

  It means architecture that is referred as a target in development. In single development, architecture affects on the quality attributes of the products, such as realization of the functional and nonfunctional requirements. In PLD, architecture affects the reusability of products, in addition to the abovementioned quality attribute. If the development of product family last to a long period, requirements to the architecture may vary in the long period. Change of reference architecture cause a big impact on cost, because it is a basis of reusability in the product scope. Reference architecture is explicitly described as an architecture document. Architecture document contains description of the role of subsystems and the dependency between subsystems.

- Implemented Architecture

  It means architecture as an abstracted structure of the implementation realized by source code. Because the implementation can not be directly compared to the reference architecture, we need the abstracted expression of the implementation that is same abstraction level to the reference architecture, in order to check discrepancy to the reference architecture. If the implementation is faithfully implemented to the reference architecture, there ought to be no discrepancy between reference and implemented architecture. To extract implemented architecture from the implementation, we can use reverse engineering from source code. Several tools for reverse engineering are available such as imagix [17] and lattix [27] [43]. Although it is hard to extract up to the original design intention that lies as a background of visualized structure by using these reverse engineering tools, they are useful to visualize internal relationship between constituent elements to understand the

abstracted structure of the implementation.

- Implementation

  It means substantial artifact such as source code and folder structure. Regardless of PLD or non-PLD, the implementation is expected to realize faithfully to the reference architecture. Although the implementation is realization of the reference architecture, the abstraction level of implementation differs from the reference architecture. For example, information of role of subsystem and the dependency between subsystems are not directly expressed on the implementation itself. Therefore the implementation can not be directly compared with the reference architecture. Implementation is explicitly described as source code of the directly structure.

## 2.3 Product Line Development

In this chapter we describe positioning of software architecture in PLD.

### 2.3.1 PLD and Architecture

Software product line is defined as “a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way,” by Clements and Northrop [9]. As indicated in [9], PLD is effective for product developments that have many common features. Generally in PLD, common architecture is used for a product family to enable large reuse in practice. Differences of architecture in PLD from non-PLD are as follows:

- Architecture centric development

  In PLD, so-called product line or reference architecture (PLA) plays a key role throughout the development. To enable large reuse in product family development, we apply the same architecture commonly for the product scope. Software architecture not only affects the structure of software itself, but also affects the organization structure of the project. PLA is also used for communications during stake holders in the projects. In non-PLD, architecture is mainly used to express software structure inside development team, in many cases.

- Variability management

  In PLD, common parts and variation point is clearly defined in the architecture. In non-PLD, variation point is not always defined in the architecture.

- Explicit product scope

  In PLD, there is explicit product scope. Architecture is decided to realize all changes within the product scope. In other words, we can optimize architecture for the product scope. In non-PLD, product scope is not always clearly recognized.

### 2.3.2 PLD in Embedded Software Development

Generally PLD is suitable for development that the features of the products are predictable to a certain extent. As an example, software development for some consumer products matches the condition.

In addition, recent these software developments face problems of increase of development scale and shortening of development time, together with product quality. Development scale problem includes increase in number of products and number of functions in each product. Increase in number of products brings needs of concurrent development of product family. Increase in number of functions brings needs of appropriate variability management.

As for shorting development time, it is mainly caused by shortening of products release cycle. In products field appearing in market, product release cycle is not established as a custom, release interval is relatively long. As the market grows, consumer needs become high, number of participant at market increases.

PLD is expected to solve abovementioned problems, and actually it has become widely used in such software development.

In products field that PLD is regarded effective, it is common that at least several similar products are already released in market. This means that there exists an asset such as source code for those similar products. In this situation, starting PLD with existing asset is one of realistic plan, rather than a Greenfield scenario [9].

## 2.4 Difficulty of Maintaining Architecture

In this section, we describe difficulty of maintaining architecture that arises in products development from several viewpoints.

### 2.4.1 Changes of Environment

Generally, software architecture is determined to incorporate considerable changes during the architecture is used. Especially in PLD, software architecture is assessed by several evaluation methods so as to incorporate changes in the product scope. However, any product line needs to evolve and adapt over time to incorporate new customer requirements and new technology constraints [44]. This is because not all changes are predictable before starting developing products.

As a result of those, architecture becomes old-fashioned in realizing new changes.

### 2.4.2 Architecture Deterioration

Architecture deterioration is a problem that has been investigated for long time. It means deterioration of implementation quality that is caused by incremental changes, fixing defects or optimizing quality attributes. These phenomena of such architecture deteriorations are known as architectural drift [38], software aging [37] or architecture erosion [18], or design erosion [19].

If all future variations are predictable, architecture deterioration by product development will not occur. However, practically deterioration occurs in some extent, because of the variations that was not forecasted at the time of determining architecture.

### 2.4.3 Architecture Gap

Architecture gap is also known problem that happens along with product development [36]. It means discrepancies between predetermined architecture and the implementations. There are two origins that may cause architecture gap in PLD.

- Gaps along with the development

  These are architecture gap that occur along with the progress of product development. This is a kind of phenomenon that is known as architecture erosion or design erosion. Ideally architecture gap is fixed as soon as it appeared. However it is difficult to fix when it appeared almost the end of development of the product because of the risk of change.

- Created gaps in reactive approach

  As an approach of composing product line assets, there is reactive approach. In reactive approach, core assets are prepared stepwise from the place that became necessary. If we change the reference architecture to realize product line assets from the existing implemented architecture, the architecture gaps between the implemented and reference architecture are created.

In both cases, immediate correction of architecture gap is not realistic, because of the risk of side effects by change and limitation of the allowed cost. Before bridging the architecture gaps, we need to aware the characteristics of the reference architecture and the implemented architecture, together with the characteristics of the gap between these two architectures.

## 2.5 Utilizing Existing Artifact

As mentioned in previous section, Greenfield scenarios [9] are seldom found in industrial contexts. It means that utilizing existing artifacts such as source code and documents is a key for saving development time and cost in creating product line core assets. In such development, needs of architecture evolution often arise.

In case of new products field development, because products continuity from existing artifact is relatively low, it is conceivable to redesign architecture even if we utilize existing artifact.

In case of similar development of products, there would not be necessary to change architecture, if requirements for the new products are almost same from products realized on existing assets. However market requirements for functional and non-functional feature become usually higher than former similar products, in most cases. In order to fulfill new requirement for the products, sometimes big change on the architecture becomes necessary. That's why architecture evolution is necessary. In this case product scope is not always clearly aware, major function of the products is kept before and after the architecture evolution. In other words, there is strong continuity over architecture evolution. Utilizing existing artifact in creating product line assets can be effective because of the continuity of products, in this case.

## 2.6 Architecture Refactoring

In this chapter, we describe issues related to architecture refactoring.

### 2.6.1 Evolution and Refactoring

From the viewpoint of maintainability and transportability of the software, refactoring is known as a technique to change the internal structure of the program without changing the behavior of the system. In this study, we call similar approach to software architecture as "architecture refactoring".

Architecture refactoring means architecture evolution to improve quality characteristics such as the maintainability and transportability with keeping the main function that is expected to be realized on the architecture. The architecture evolution to cope with a technical and business environmental change may be accompanied by the expanse of a function assumed on the architecture. It can have the side that is slightly different from the refactoring of the source code that does not to change a function.

However, we deal with the situation to keep a continuity with the past product group, and we don't consider it until the change of an essential and large assumption function, in this study. Therefore we use the term "refactoring" as a part of evolution

aiming the improvement of the quality characteristics for assumption functions.

### 2.6.2 Refactoring in Former Study

The characteristic in a source code known as the sign which seem to cause a problem in the future observed at the time of software development from experience is called “Bad smell”. Fowler [13] mentions 22 kinds of bad smells such as “Duplicated code” or “Long method”. (see appendix B)

Although concerns of smells and the measures by Fowler are closed to the source code, the idea of “Bad smells” can be extended to architecture level. Rooks [41] propose idea of “Architecture smells” in order to utilize a trigger for big refactorings, where they observe structure smell such as cyclic dependency or concentration of functions on several architecture levels, such as package, subsystem, and layers. Regarding to architecture smells, Pollack [40] also mentions five architecture smells such as “Business logic is tightly coupled to non-functional requirements”.

The methods for using an architectural smell to point out the need for refactoring are described in [41][14]. These methods utilize the structural information of the implemented architecture. Fenton et al. [12] suggest an approach to identify refactoring needs by using software metrics. Metrics point out problems objectively, but the difficulty is that the use of such metrics does not always point out the real need for refactoring. This commonly takes place with “code smells,” as discussed in [13].

Each of them is consideration for the implemented architecture, and it is not enough to plan a strategy for refactoring that involve the reference architecture in PLD.

### 2.6.3 Impact on Cost and Implementation Quality

There are a lot of cases to be accompanied by a large-scale change of the implementation, and the change of the architecture entails a modified cost generally. As an example, we consider a case of changing functional decomposition between subsystems. In this case, because changes occur in each related subsystem, cost for change increase along with the number of change increases.

Furthermore, the quality risk by the change exists, too. Because a change point is not closed in a specific module and subsystem by the change of the implementation with the change of the architecture, the range that should confirm influence is wide in comparison with the refactoring closed to a specific module.

Thus, because cost and influence on quality are big by the architecture refactoring, we need enough assessment for influence range.

### 2.6.4 Difficulty of Architecture Refactoring

The difficulty in refactoring PLA lies in determining an appropriate refactoring strategy by considering the refactorings for both the reference architecture and the implemented architecture. The term “architecture refactoring” refers to a large refactoring at the architecture level. In [41], “large refactoring” is described as the composite of small refactorings that improves detailed design and code quality. The basic behavior of the outside of the system does not change before and after these architecture refactorings. In refactoring PLA, we have to consider both the reference architecture and the implemented architecture. The term “reference architecture” here refers to the planned and targeted architecture. The term “implemented architecture” refers to the architecture that is realized as the implementation.

In several product line practices [23][26], there are migration approaches of the implementation toward the reference architecture. In these cases, the implemented architecture is refactored in order to adjust the deviation from the reference architecture.

To identify the refactoring needs for the implemented architecture itself, a method of utilizing “architecture smell” [41] is available. Although this is useful to determine the refactoring needs easily from the structure of the implementation, we need further examination from the viewpoint of PLD in order to apply the abovementioned method to PLA because architectural smells are based on the implementation at a certain point of time.

With regard to the reference architecture, the objective of refactoring is to increase the productivity and quality of future products, such as the ease of realizing future variability and the testability of products. If a product line lasts for a long span of time, sometimes a reengineering of the architecture becomes necessary in order to realize an innovative feature to adapt to increasing market requirements. In order to optimize the reference architecture, we might have to approach reengineering. However, in a large software project, designing from scratch is often not realistic. In such a case, refactoring from the existing architecture is rather useful. Thus there are different aspects to reference architecture and implemented architecture, and we need to distinguish between them for refactoring PLA.

Once the reference architecture is refactored, the corresponding implementation needs to be refactored as well. When the latter procedure needs time to follow, there appears a gap between the implemented architecture and the reference architecture until the refactoring on the implementation is completed.

Although there are various refactoring needs that range from the reference architecture to the implemented architecture, performing architecture refactoring ad hoc induces confusion into the project. In order to conduct these refactorings effectively, we need to carry out PLA refactorings under appropriate prioritization by considering

the related metrics result, the influence and cost of refactoring, the situation of the project, business objectives, and so on. We propose the means of doing so in this paper by using a problem factor portfolio analysis that we explain in chapter 4.

### 2.6.5 Refactoring Needs in PLD

In PLD, it is important to determine product scope before preparing product line core assets. However, even in the product scope, it is often found to update domain artifacts for adapting new variability. In other words, small scale of architecture refactoring is often taken place during the product scope.

In addition, we consider about composing product-line assets phase. In proactive approach of PLD [20], reference architecture is defined prior to compose core assets, so as to be used commonly for products within the product scope. Generally in PLD, because same architecture is used for relatively long time compare to single system development, architecture refactoring is sometimes necessary to adapt following situations:

- Technology improvement: Major technology improvement may occur during the period of product line scope.

- Unexpected requirement from market: Market requirements are always changing. Sometimes market requirement that can not be expected at the beginning of product-line scope.

In reactive approach of PLD [10], core assets are composed stepwise by using existing assets. It means that at least the related part of the architecture is refactored in order to composing core asset.

As we mentioned above, architecture refactoring is necessary in PLD, whether the approach of composing product-line core asset is reactive or proactive. Architecture refactoring is not only necessary, but also an essential technique in order to lengthen the life of PLA.

### 2.6.6 Refactoring of Reference Architecture

In considering architecture refactoring, changes occurs in both implemented and reference architectures. In addition, there can be discrepancy between them, in products development project. Therefore, we think that we can make problems clear by considering problems belong to reference and implemented architecture separately.

Figure 2 expresses the relationship between the reference architecture and the implemented architecture. The implementation is refactored so as to resolve the

discrepancies between the reference and implemented architecture, and the reference architecture is refactored so as to fulfill new requirements.

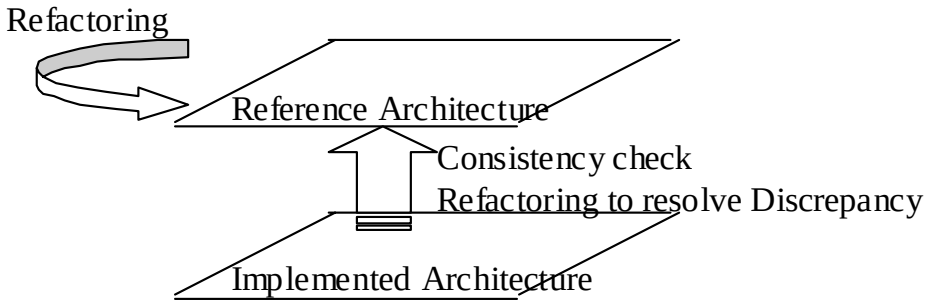


Figure 2 Relationship of Reference and Implemented Architecture

Refactoring of reference architecture is change of rule that restrain software design. When these changes cause discrepancies between reference architecture and the implementation, it is recommended to change the implementation to follow the reference architecture, as soon as possible. Those discrepancies that is caused by the change of reference architecture is caused by changes of the rule between components, it might need a big impact on the implementation.

As Roock points out, large change of implementation in subsystem level induces a lot of cost. Change of reference architecture induces more cost and influence to the project. In general, there are various candidates of refactoring at a certain moment of development, therefore effects and influence to the project depends on choice of refactoring from candidates and timing of performing them. We treat this as an issue of decision taking, and we propose a technique in this study.

### 2.6.7 Refactoring of Implementation in Architecture Level

Refactoring of implementation in architecture level means large scale refactoring which affects overall structure of the software. Such large refactoring on the implementation is a combination of each small source code refactoring. Although each small refactoring is same as fowler's refactoring, the difference from small scale of source code refactoring is its scale or number of combination. Roock [41] mentions that we need different attention in addition to regular refactoring that Fowler [13] describes. For example, consideration of refactoring timing and preservation of working hours on the products' project belongs to them. However, almost all examples of large refactoring by Roock focus on refactoring on the local structure such as moving class or change of inheritance. To solve architectural problems at subsystem level, we need other viewpoints and techniques that consider relation to the reference architecture.

## 2.7 Refactoring Opportunity

Figure 3 shows the process flow in cyclic product development after the application of the abovementioned method. Smells are projected to a refactoring item by considering the problem factors. The derived refactoring items are assessed using problem factor portfolio (PFP) that we explain in section 4.4.5 . The assessment here refers to the process that determines the position of each refactoring item in the PFP plane and the effort required for the item. Refactoring items are then selected after a consideration of the magnitude of the problem factor, the effort required, and the situation of the project. Refactoring is then performed on the selected item, and the rest are assessed at the next assessment opportunity together with new refactoring items derived from new bad smells that we explain in section 4.3.1 .

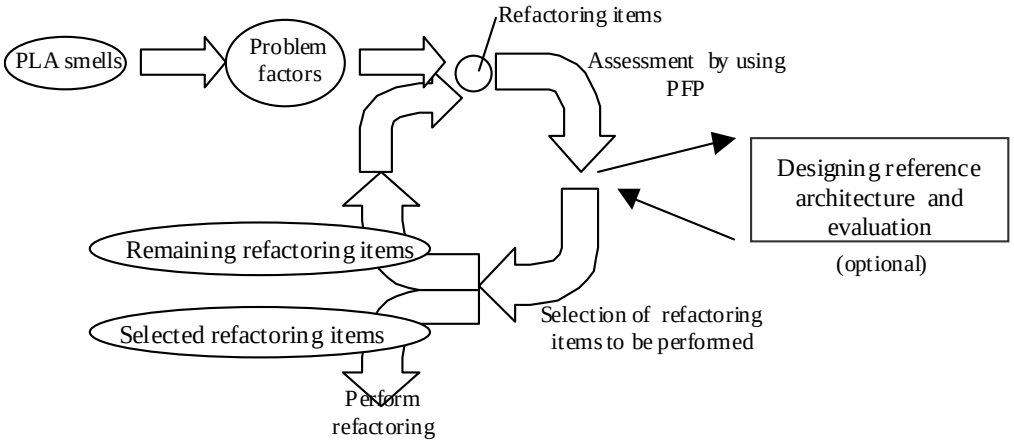


Figure 3 Process Flow in Cyclic Product Development

# Chapter 3

# Problems

In this chapter, we describe the problems that we are going to solve in this study. In section 3.1, we explain the main problem that we have to solve. In section 3.2, we describe the subsidiary problems as practical techniques to solve the main problem.

## 3.1 Main Problem

We set the objectives of this study is to provide a technique for architecture refactoring as follows;

- A technique to support decision taking of prioritizing architecture refactoring items

In the development of family products in product-line context, it is essential to select architecture in consideration of overall optimization of product development in the product scope.

As techniques for selecting architecture, ATAM, SAAM, Pulse-DSSA and such are known. These techniques are useful for selecting architecture at the early stage of development, because they judge the good or bad of architecture based on requirement analysis, without considering the continuity with the existing assets. In PLD, we often encounter the needs of architecture refactoring to adapt new requirements, because PLD ranges for long time compare to non-PLD development. In considering architecture refactoring, we have to consider the continuity with the existing assets.

As techniques for refactoring, methods of Fowler [13] or Roock [41] are known. Fowler describes the detailed techniques for detecting refactoring needs and specific refactoring techniques in source code level. Roock proposes similar techniques to Fowler's that handle large scale refactoring. In architecture refactoring in PLD, treatment of reference architecture is important, because it regulates the relationship among the components for large reuse. Although these techniques cover refactoring of implementation, they do not cover the relationship between the reference architecture as the design intention and the implemented architecture as an abstracted expression of the implementation.

Because architecture refactoring need high cost, generally it is hard to deal with all candidates of architecture refactorings that we thought of. Therefore we have to select effective items under the limited resource of cost for refactoring. However there were not appropriate technique to select effective refactoring items, we have selected them ad-hoc based on the past experiences.

## 3.2 Subsidiary Problems

In order to solve the main problem described in previous section, we found subsidiary problems as practical techniques. We explain those subsidiary problems as follows:

- Detecting problems related to development of series products

  In large software project, it is hard to specify the bad symptoms to be improved, because of complexity of the software itself and the complexity of the development process.

  As a technique for detecting bad symptoms, using "smells" is known in Fowler etc. However known "smells" are based on the structural observation at a moment, it is difficult to find out problems that come from development of series products. Our objective related to detecting problems is to provide techniques to find out problems that lie on plural development of products.

- Quantification of the magnitude of problems

  There are various kind in problems related to architecture. Generally it is considered to be difficult to compare different kind of problems because the metrics means for each problem differ. Therefore the magnitude of the problems has been judged only subjectively. Our objective for handling magnitude of problems is to make possible to compare different kind of problems.

- Prioritization of refactorings

  However there can be many refactoring candidates depending on projects, we don't have appropriate to means to judge the order to deal with.

  In taking decision on refactoring, many factors should be concerned, because architecture refactoring affects big impact on quality and development cost. These factors involve the extent of problems being solved, cost and risk for the refactoring, and future product plan. Our objective for prioritization of refactoring is to provide useful materials for decision based on metrics.

# Chapter 4

# Proposed Technique

In this chapter, we propose a technique for decision taking method for architecture refactoring, in which we consider the background and problems we discussed in chapter 2 and 3. In section 4.1, we explain requirements for the technique. In section 4.2, we outline the proposed technique. In section 4.3, we explain fundamental ideas that are necessary to describe our technique. In section 4.4, we explain the detail of the technique.

## 4.1 Requirements for the Technique

Because an influence range is wide and its cost is high as for architecture refactoring, the misjudgment for refactoring brings big damage on the project. Therefore, we need to be careful in planning architecture refactoring. In addition, technique for architecture refactoring should fulfill appropriate requirements. From this viewpoint, by referring subsidiary problems mentioned in section 3.2, we set following three points as important requirements for the decision taking technique of architecture refactoring. We propose technique to satisfy these in this study.

- Based on observed data

  The design of the software architecture is often carried out based on the intuitive judgment by the expert. When there are not enough data becoming the grounds of the architecture decision, such as in case of designing a new type of product line, those intuitive judgment is often useful. However, because those judgments are not based on explicit knowledge, sometimes it is difficult to judge the authenticity of the judgment objectively.

  In case of architecture refactoring, we can observe facts to become the motive and the grounds of the architecture refactoring from the data of the existing project. We utilize such data by the proposed technique and aim at providing objective technique.

- Targeting common problems that affect most products in the project

  In PLD, we develop family products under the common architecture. Problems due to the architecture cause a similar problem to the other products in the product scope.

By solving the problem that was common to the project group, similar effect can be expected in a project of afterward. Therefore, a big effect is expected for the whole projects, by solving the common problem.

- Providing quantitative information for taking decision

  Generally, cost and time that is allowed to refactoring is limited, we need to prioritize refactoring considering the situation of the project. Here the situation of project means the factor of allowed cost and time, variations of products of afterwards, and the number of products planned. For example, if there are several refactoring candidates that have similar cost and effects, we can decide the priorities among them according to the situation of the project. In addition, we can take a decision of leaving a refactoring plan, even if it is really cost effective, by considering the situation of the project. For such decision taking, we need quantitative information regarding to the problems.

## 4.2 Overview of the Technique

Proposed technique comprises 7 steps. Followings are summary of the steps in proposed technique.

**STEP 1** Select bad smells

Based on the experience of products development by using common infrastructure, we detect problematic phenomena that seem to obstruct the smooth execution for product development.

**STEP 2** Find problem factors for bad smells

By analyzing bad smells that are gathered in STEP1, list up the origins that cause those bad smells. We list up the origins as problem factor (see section 4.3.2 ). Then group them into problem factors that are related to the reference architecture and the implemented architecture.

**STEP 3** Determine refactoring items

Plan refactoring items to solve the problem factors got in STEP2. In addition estimate the effort needed for the refactoring.

**STEP 4** Quantification of the magnitude of problem factor

Quantify the magnitude of problem factor identified in STEP2. Problems are quantified by using appropriate metrics for each problem factor. In order to compare the result of different metrics, we normalize the result according to the five-grade system to express the degree of annoying developers. The boundaries of

five-grades are predetermined from project's experiment.

STEP 5 Portfolio analysis

Plot normalized magnitudes of problem factors on PFP plane. Then grasp the trends of problems for reference of decision taking to deal with them.

STEP 6 Judge priority of refactoring

Judge priorities of refactoring items of refactoring under total consideration of refactoring efforts obtained in STEP 3, the magnitudes of problem factor obtained in STEP 4, trends of bad smells plotted in STEP 5, and current situation of project.

STEP 7 Execute refactoring

Execute refactoring according to the judgement in STEP 6.

## 4.3 Fundamental Ideas

We explain general ideas that are backgrounds of our approach. In section 4.3.1 we explain bad smells in this study by comparing with bad smells that is already known in former study. In section 4.3.2 we explain problem factors which are structural reasons inducing bad smells. In section 4.3.3 we explain refactoring items which are measures for those problem factors.

### 4.3.1 Bad Smells

In our proposed technique, we define bad effects to the cost and quality that is continuously observed during architecture related activity in development of product family as "bad smell". Here the activity means the achievement of addition, change of the function and non-functional requirement and quality properties.

In former study such as Fowler [13] and Roock [41], bad smell is based on structural observations of the implementation, as we mentioned in section 2.6.2 . Because they are based on structural observation at a moment of development, we need another technique to detect bad symptoms that is specific to the development of product family such as PLD.

These bad smells, of course, involve various issues that include architecture, personnel, and management. In this study, however, we confine ourselves to the bad smells that originate in architecture. In addition, we call those bad smells that originate in architecture as "PLA smells". We ascertained PLA smells can be categorized in S1 through S3 as follows.

- Influence to cost for change of implementation (S1)

  Changes on the implementation are inevitable in development. Followings are bad smells on cost that is induced by the change.

  Ex.1) Large man-hour for inspecting influence by the change or adding functions
  Ex.2) Need long time to find out change point in a specific huge subsystem

- Influence to quality (S2)

  Whenever we change the implementation, there are risks on quality. Followings are bad smells related to quality of products.

  Ex.1) Relatively large numbers of bug are reported in comparison with other subsystems.
  Ex.2) Similar bugs found repeatedly

- Influence to the cost for verification(S3)

  In addition to development cost, cost for verification occupies not negligible in total development cost. Test cost often depends on the structure of implementation of software. Followings are those examples

  Ex.1) Too many combination in test
  Depends on the number of branch, we should prepare test case.
  Ex.2) Too many number of regression test
  Large set of regression test is always necessary, when we can not specify the influence range in a small range.

### 4.3.2 Problem Factor

We call architectural cause of bad smells as “problem factor” in this study. Generally bad smells originate in various causes, such as software structure, technical skills of developer, or development processes. Among them, we pay attentions on the cause related to software architecture. In this section, we explain the kinds of problem factor and magnitude of problem factor.

#### (1) Kinds of Problem Factor

In order to find out problem factor, we need to investigate artifacts such as source codes and architecture documents in detail. Architecture documents include description of software structure such as role of subsystems and the relationship between subsystems. To investigate these artifacts, hearing to the development engineer facing a problematic phenomenon on a daily basis is effective.

Problem factor can be divided into reference architecture related and implemented architecture related. We ascertained problem factors can be categorized in Pr1 through Pr2 for reference architecture and Pi1 through Pi3 for implanted architecture. For perspective, we refer to code smells and architecture smells known for an existing study.

**Problem factors due to the reference architecture**

Problem factor due to the reference architecture are mainly related to inappropriateness of roll assignment of subsystem. Inappropriateness can be categorized into concentration and desperation.

- Concentration of functions (Pr1)
  Followings are typical example of concentration of functions.

Ex.1) Role assignment for specific subsystem is too wide, so the code size of the subsystem is large.
Ex.2) Dependencies from other subsystem is concentrated to specific subsystem.
Ex.3) Too many global access

- Dispersion of functions (Pr2)
  Followings are typical example of dispersion of functions.

Ex.1) Inappropriate assignment of role
Ex.2) Too many dependencies between subsystems

**Problem factors due to the implemented architecture**

Problem factor due to the implemented architecture can be categorized into following thee categories.

- Kinds of implementation techniques (Pi1)
  This category of problem factor is caused by the selected techniques for implementation.

Ex.1) Maintainability problem because of too many compilation switches

- Quality of the implementation (Pi2)
  This category of problem factor is caused by the quality of implementation.

Ex.1) Long functions
Ex.2) High complexity
Ex.3) Considerably complicated relationship between components

- Alignment to the reference architecture (Pi3)

  This category of problem factor is caused by the gaps between planned and actual architecture.

Ex.1) Different contents that compose the subsystem
Ex.2) Discrepancy in dependencies
Ex.3) Misplacement of component

The term "component" refers to an element that composes an asset such as a function, group of functions, class, package, or subsystem. In many cases, a PLA smell is not related simply to only one category because the problematic symptom is often multifaceted. For example, when there are difficulties in fulfilling a certain requirement, the problem often lies in both the architecture and the complexity of the implementation.

When a project has reference architecture, discrepancies between the reference architecture and the implemented architecture can often be observed. In the practice involving several product lines [23][26], a migration approach is used in the implementation of reference architecture. In such a case, a planned refactoring of the implementation is important because the implemented architecture reflects the original difference from the reference architecture.

Even if there is sufficient consistency between the implemented and the reference architecture, the implementation is in the danger of deteriorating if its consistency is not checked regularly [36]. Several tools that are useful in detecting deviations in the architecture's implementation have been reported [11][43].

On the other hand, the reference architecture is also in danger of becoming outdated. To adapt to the new requirements of the product or to make its development more sophisticated; it then becomes necessary to refactor the reference architecture. For PLD, it is important to optimize refactoring by considering the reference-architecture related and the implemented-architecture related problem factors.

### (2) Magnitude of Problem Factor

For quantitative comparison of problem factors, we need to measure the magnitude of each problem factor. Using metrics is an objective method for measuring problem factors quantitatively. We should choose metrics to meet each problem factors. Table 1 shows an example of metrics and problem factor that is used in our project based on experiences of product development.

As for metrics of the reference architecture, there are two approaches to measure. First one is direct method that measures the magnitude of reference architecture, such as scoring on architecture document. Second one is indirect method that measure

through the implementation deliverables including implemented structure and the source code that reflected design rationale such as the folder constitution. However the direct measurement seems to be ideal, there are pros and cons for each approach as follows:

- Direct method

  Although this method can perform architecture evaluation directly to the reference architecture, the result is not always accurate because of inaccuracy comes from description of architecture documents that is written in natural language and inaccuracy of scoring method that is based on the intuition such as feeling or impression by the human who evaluates the architecture.

- Indirect method

  Although this method has intrinsic inaccuracy that is based on the discrepancies between the reference and implemented architecture, there is no inaccuracy that lies in direct method as abovementioned. Under the condition that the implementation is done almost close to the reference architecture, and the characteristics obtained from metrics of is enough remarkable, we can use the indirect method for measuring the magnitude of the reference architecture.

Table 1 is an example of choice of metrics in case of the second approach mentioned above. In this example, as a result, the number of dependence is used by both evaluations of reference and implemented architecture, but we perform the measurement at a point reflecting a factor in each structure. About concrete values at variable expression in Table 1, we predetermine based on the experiences on the target project in advance. By normalizing each metrics in 5-step, we can compare the magnitude of different problem factors.

Table 1 Classification for Magnitude of Problem Factors

| Problem factor | Problem Located | Metrics | 5 Degrees of Evaluation Small ←(Magnintude of Problems)→ Large | | | | |
|---|---|---|---|---|---|---|---|
| | | | 1 | 2 | 3 | 4 | 5 |
| Broad Responsibility of Subsystems | Reference | LOC of the sybsytem | <(L1) | <(L2) | <(L3) | <(L4) | (L4)≦ |
| Dependencies from everywhere | Reference | Number of dependencies | <(N1) | <(N2) | <(N3) | <(N4) | (N4)≦ |
| Too many dependencies | Reference | Number of dependencies | <(N1) | <(N2) | <(N3) | <(N4) | (N4)≦ |
| Too many compilation switches | Implemented | Dencity of Compilation Switches | <(R1)% | <(R2)% | <(R3)% | <(R4)% | (R4)%≦ |
| Long functions | Implemented | Average LOC per functions | <(L1) | <(L2) | <(L3) | <(L4) | (L4)≦ |
| High complexity | Implemented | Avarage of Cyclomatic Complexity | <(C1) | <(C2) | <(C3) | <(C4) | (C4)≦ |
| High connectivity | Implemented | Number of dependencies | <(N1) | <(N2) | <(N3) | <(N4) | (N4)≦ |
| Different components | Implemented | Numbe of different components | <(F1) | <(F2) | <(F3) | <(F4) | (F4)≦ |
| Discrepancy in Dependency | Implemented | Number of dependencies | <(N1) | <(N2) | <(N3) | <(N4) | (N4)≦ |

### 4.3.3 Refactoring Items

The term “refactoring items” refers to an individual refactoring plan to resolve the corresponding problem factors. Because factoring items corresponds to problems of architectural level, its granularity is larger than regular source code refactoring. Whereas regular source code refactoring indicates refactoring actions on specific portion of codes, refactoring items generally involves plural refactoring actions. In other words, it is a kind of policy for refactoring to solve the problem.

**Refactoring items corresponding to reference architecture**

Refactoring items for reference architecture are planned from the viewpoint of improving functional decomposition. We ascertained refactoring items corresponding to reference architecture can be categorized in Rr1 through Rr3 as follows.

- Splitting or creating subsystem (Rr1)

  Refactoring items in this category are that increase the number of subsystems.

Ex.1) Separate portions that is related to specific concern

Ex.2) Create new subsystem to prepare realizing new functions

- Merging subsystems (Rr2)

  Refactoring items in this category are that decrease the number of subsystems.

  Ex.1) Gather similar subsystems into one subsystem
  Ex.2) Extend the role of subsystem to include new functions (assume that once assign new subsystem for new functions, then absorb it into existing subsystem)

- Change role of subsystems (Rr3)

  This is the case that the number of subsystem does not change before and after the refactoring.

  Ex.1) Change functional decomposition between subsystems

<u>Refactoring items corresponding to implementation</u>

Refactoring items for implementation are planned from the viewpoint of improving functional decomposition.

Refactoring item involves actions of modification to the implemented source code, such as “Move package A from layer X to layer Y,” “Split package B into packages P and Q,” or “Merge module A and module B.” Practical operations in a source code are the same as general source code refactoring, but they differ from general source code refactoring in that these refactoring items affect the software architecture. We ascertained refactoring items corresponding to reference architecture can be categorized in Ri1 through Ri2 as follows.

- Refactoring involving changes of implementation technology (Ri1)

  Refactoring items in this category changes a way of implementation to make it more sophisticated.

  Ex.1) Change realization technique of variation point
  As realization techniques, three timing is known. These are compilation time, binding time, and execution time. Changing techniques for realization means change of timing. As an example, there is a method to change from compilation switch to binding options.
  Ex.2) Virtualization of procedure by applying function pointer

- Refactoring without changing implementation technology (Ri2)

  Refactoring items in this category improves quantitative aspect of the implementation.

Ex.1) Split long functions into small ones
Ex.2) Shallow condition nest
Ex.3) Remove unused compilation switches
Ex.4) Move implementation between subsystems.

### 4.3.4 Relationship among Fundamental Ideas

Generally relationship between bad smells, problem factor and refactoring items are many-to-many as illustrated in Figure 4. This means, there are multiple problem factors that cause a bad smell, and we need to combine several refactoring items to deal with these problem factor.

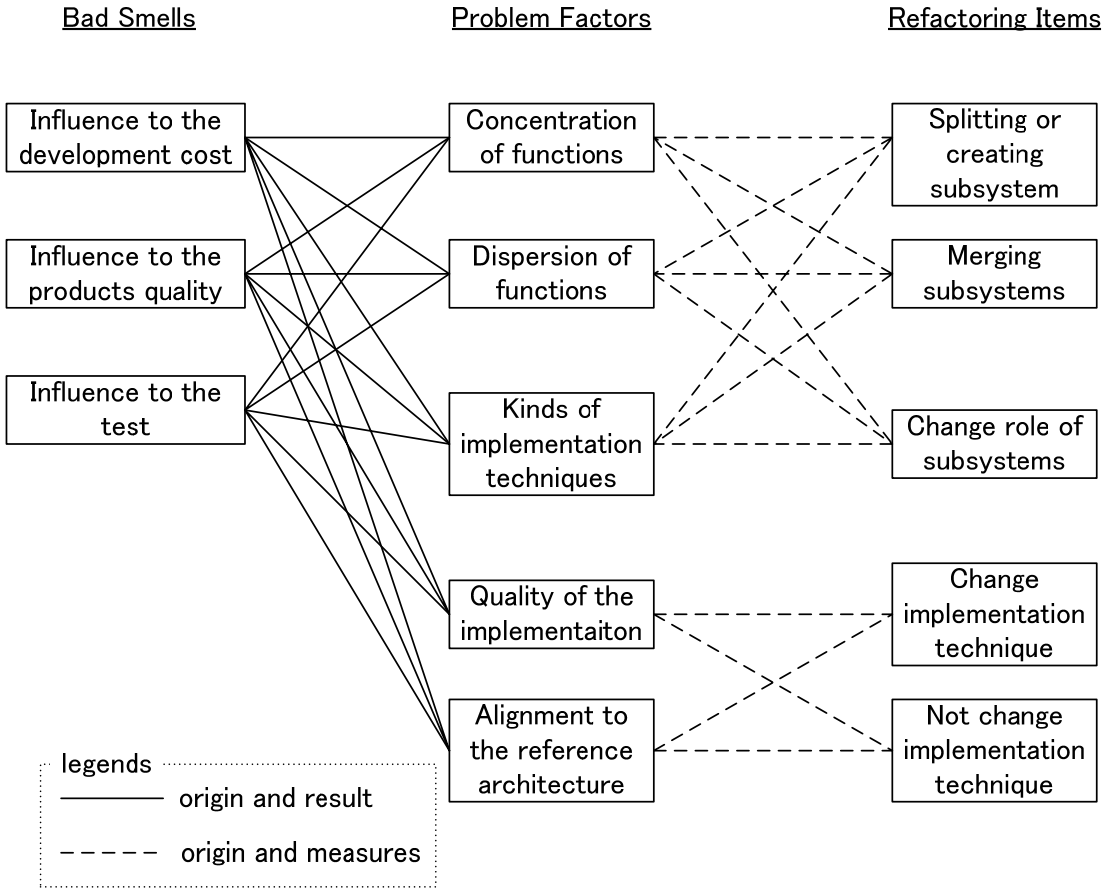


Figure 4 Relationship between Fundamental Ideas

## 4.4 Steps of Proposed Technique

In this section, we describe detailed process of each step in the proposed technique. Figure 5 shows an overview of the steps. In Figure 5 arrows connecting each step means process order and delivery of information. For example, in step 1 we select bad smells from bad effects observed in the projects, then the bad smells are used as an input for step 2, and so on.

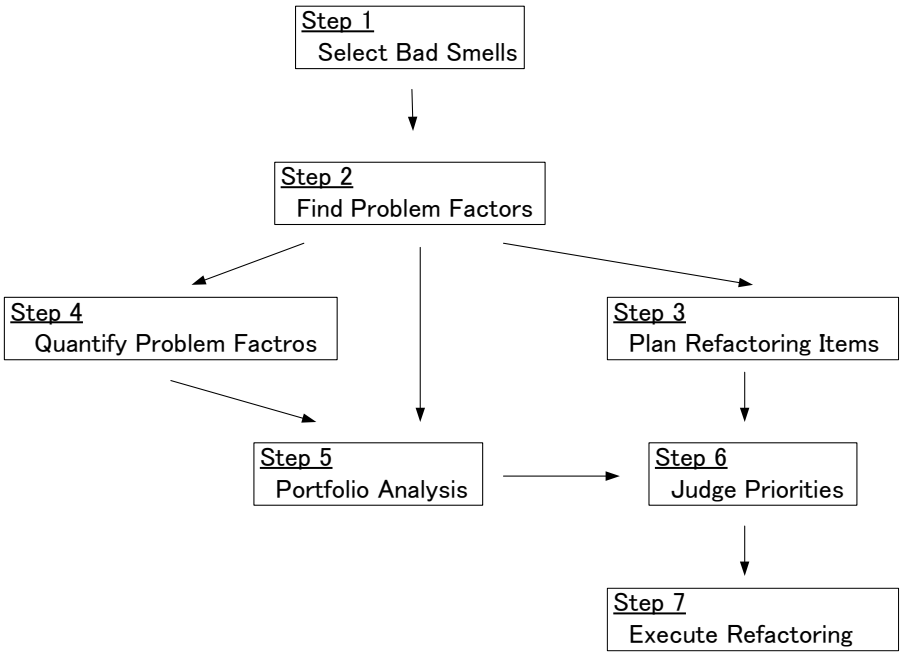


Figure 5 Overview of the Steps

### 4.4.1 STEP 1: Select Bad Smells

In this step, we find out bad smells from the experience of products developments. Table 2 shows outline of STEP1.

Table 2 Outline of STEP1

| Items | Descriptions |
|---|---|
| Objective | Sorting out bad smells from project data |
| Input | Bad effects observed in projects of product development |
| Output | List of "Bad smells" |
| Procedure | 1. List up bad effects from the project data<br>2. Pick up bad smells from bad effects |
| Note | Viewpoints for finding bad effects are production cost and quality of the products.<br>Pick up bad effects that emerge in multiple products as bad smells. |

In order to find bad smells, first we collect bad effects on development that seem to obstruct smooth execution for a product development. Then we repeat collections of bad effects for several products. Viewpoint to collect those bad effects are follows:

- Production cost for development
- Quality of the products

After we collect these bad effects for several products, we select bad effects that appear commonly in several products. We call those bad effects "bad smells" from the project in our study. Conditions for detecting bad smells are follows:

- Should appear at least two or more products.
- They may appear in future products.

In most cases, candidate of bad smells appear in two different products. However, even if they appear in two different products, when they are obviously limited to those two products, they can not be bad smells. Regarding to appearing timing, we don't care. Target products can be developed at same time, or can be developed sequentially.

Figure 6 shows an image of finding out bad smells at the project. In Figure 6, circles correspond to existence of problem. For example, product 1 has problems 1, 2, 3 and n, product 2 has problems 1, 3, 4 and n, and so on. Here we can see that problem1, 3 and n

are found commonly in several products, while problem 2 and problem 4 found at only one project. We indentify those commonly found problems as “bad smells”.

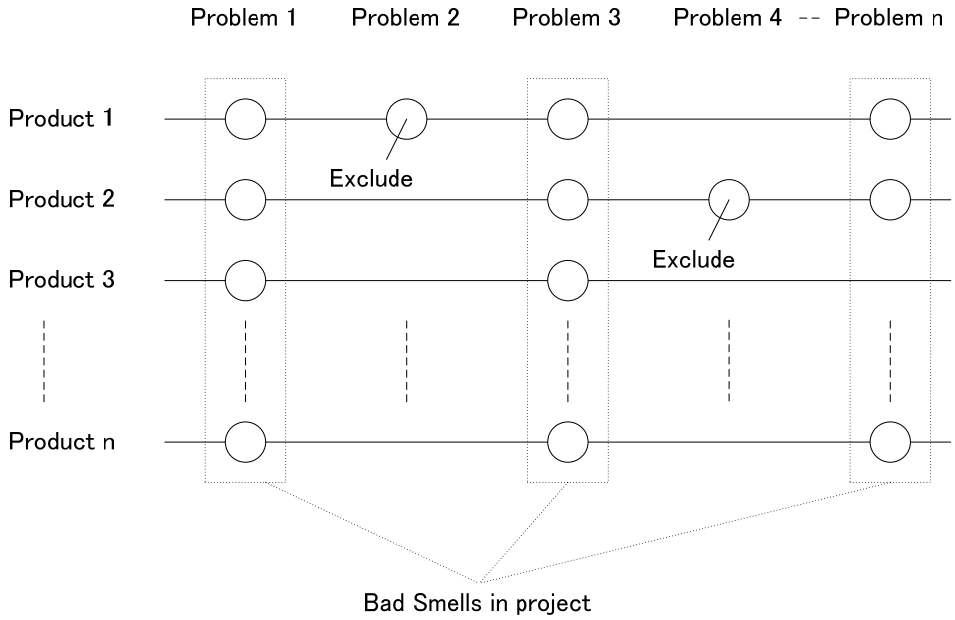


Figure 6 Image of Finding Out Bad Smells in the Project

### 4.4.2 STEP 2: Find Problem Factors

By analyzing bad smells that is gathered in STEP1, we find out the architecture-related origins that cause those bad smells. Analyzing means investigation into source code and architecture documents. Table 3 shows outline of STEP2.

Table 3 Outline of STEP2

| Items | Descriptions |
|---|---|
| Objective | Finding out problem factors for bad smells |
| Input | Bad smells |
| Output | Problem factors for bad smells |
| Procedure | 1. Seek origins of bad smells<br>2. Exclude origins that are not related to architecture.<br>3. Classify the problem factors into reference-architecture related and implemented-architecture related. |
| Note | Analyze by investigating into source code and architecture documents |

Figure 7 shows an image of filtering out problem factors that has no relationship to the architecture. In many cases, we can find multiple origins for a bad smell. Generally origins contain architecture related, skill related, and process related. We select architecture-related origins as problem factors for architecture refactoring. Architecture related means that the cause of problem is based on the structure of software or characteristics of the implementation. Here we exclude the problem factor that is not related to architecture. At this point, we also exclude bad smells that does not have any architecture related problem factor, because they can not be fixed by architecture refactoring. Examples of these excluded origins are, such as educational problems, communication between stake holders, and development process. Off course these are also problems for the problems to be taken measure appropriately in the project. However our proposed technique does not include them as its scope in this study.

After filtering out extra problem factors, we classify the remaining problem factors into reference-architecture related and implemented-architecture related.

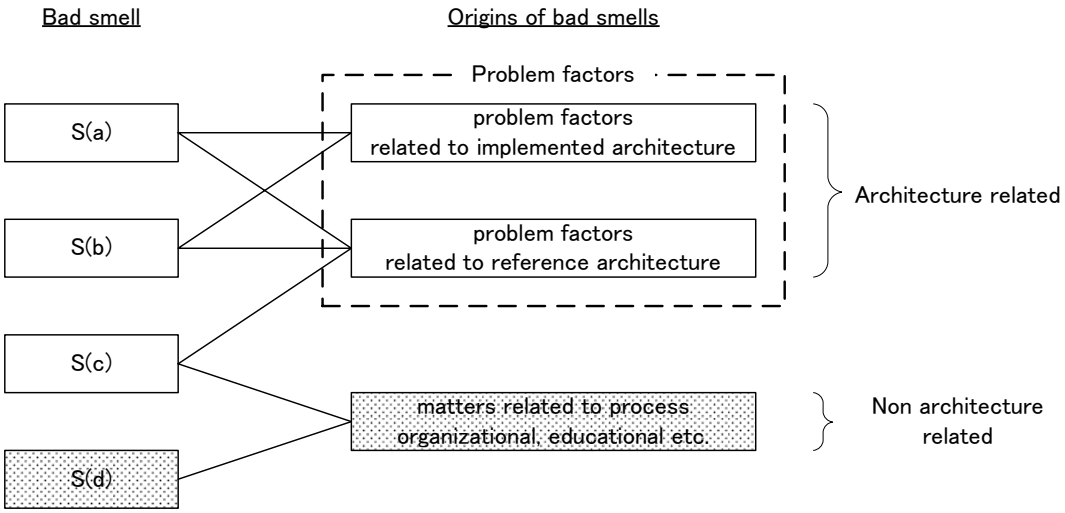


Figure 7 Image of Filtering Out Problem Factors

### 4.4.3 STEP 3: Plan Refactoring Items

In this step, we plan refactoring items to solve the problem factors got in STEP2. Refactoring item means an individual refactoring plan to resolve the corresponding problem factor. For each refactoring item, we estimate the effort to perform the refactoring actually. Table 4 shows outline of this step.

Table 4 Outline of STEP3

| Items | Descriptions |
|---|---|
| Objective | Planning refactoring items |
| Input | Problem factors |
| Output | Refactoring items to solve problem factors<br>Effort needed for each refactoring items |
| Procedure | 1. List up conceivable refactoring items for each problem factor.<br>2. Calculate man-hour for each refactoring item |
| Note | Viewpoint for planning refactoring items is to decrease magnitude of problems.<br>Estimate effort for refactoring item by using unit man-hour for a refactoring action and total number of refactoring portion. |

**(1) Plan refactoring items**

We plan refactoring items so as to solve the problems by changing architecture. View points for planning refactoring items are follows:

- Refactoring items to decrease the magnitude of problems
- Refactoring items to improve or preserve the quality of products

If the problem factor is enough concrete, refactoring items can be found in relatively small range. However, if the definition of problem is broad, there may be several candidates for refactoring items. When there are multiple candidates as refactoring items, it is recommended to list up all of them at this moment. Depending on the problem factors, we may reach same refactoring item for different problems problem factors. It is no problem. After estimating man-hour, we choose the most effective one. The criteria for choosing refactoring item among them are follows.

- Most effective in man-hour
- Common refactoring items for different problem factor

### (2) Estimating the effort for each refactoring items

Effort means man-our or production cost that is necessary to perform refactoring on implementation. Implementation means source code, corresponding architecture document that represents software structure, and subsidiary artifacts that are necessary to compose an executable module for the target products.

Generally the total production cost can be calculated by accumulating each unit refactoring production cost. Following equation (1) shows the calculation of production costs of total refactoring.

$$Cost_{total} = \sum_{m} \sum_{n} Cost_{modify}(m) + Cost_{test} + Cost_{document} \qquad (1)$$

In equation (1), $Cost_{total}$ represent the total production cost of total refactoring. Parameter "m" represents the number of kinds of refactoring operation, and parameter. "n" represents the number of point targeted for the operation. Kinds of refactoring operation are based on the difference of actual operation. For example, splitting function, move definition of variables or functions, change interface of module, removing compilation switches, and so on. $Cost_{modify}(m)$ represents the average production cost for one target point for the kind of refactoring operation on source code. $Cost_{test}$ represents the cost for regression test for detecting side effects by the change. Depending on the project, regular product test can substitute for the regression test. Then the term $Cost_{test}$ is estimated as zero. $Cost_{document}$ means the cost to occur indirectly that is represented by maintaining architecture document. The term $Cost_{test}$ occurs in case of reference architecture refactoring. There are costs for education, communication with stakeholders, and subsidiaries as $Cost_{document}$, other than cost for documenting, too. It is convenient to classify the total production cost into grades such as H/M/L for a rough analysis.

### 4.4.4 STEP 4: Quantify Problem Factors

In this step, we quantify and normalize the magnitudes of problem factors identified in STEP2. Table 5 shows outline of this step.

Table 5 Outline of STEP4

| Items | Descriptions |
|---|---|
| Objective | Quantification and normalization of magnitude of problem factors to compare different kinds of problems. |
| Input | Problem factors<br>Classification list for magnitude of problem factors |
| Output | Magnitude for each problem factor in five-grades |
| Procedure | Quantify magnitude of problems using appropriate metrics.<br>Prepare classification list for normalization of magnitude<br>Normalize the magnitude of the problem factor in five-grades by consulting a classification list. |
| Note | Prepare classification list before consulting. |

**(1) Quantification**

We quantify magnitude of problem factors by using appropriate scoring or metrics for each problem. Guideline of selecting scoring or metric is whether it represents the magnitude of problem with objectivity. Table 6 shows examples of typical problem factors that due to reference architecture and implemented architecture respectively.

Table 6 Example of Classification List

| Problem factor | Problem Located | Metrics | 5 Degrees of Evaluation Small ←(Magnintude of Problems)→ Large | | | | |
|---|---|---|---|---|---|---|---|
| | | | 1 | 2 | 3 | 4 | 5 |
| Broad Responsibility of Subsystems | Reference | Scoring | LL | LM | MM | HM | HH |
| Broad Responsibility of Subsystems | Reference | LOC of the sybsytem | ＜(L1) | ＜(L2) | ＜(L3) | ＜(L4) | (L4)≦ |
| Discrepancy in Dependency | Implemented | Number of dependencies | ＜(N1) | ＜(N2) | ＜(N3) | ＜(N4) | (N4)≦ |

In Table 6, two metrics are shown for a problem of “broad responsibility of subsystems”. As we described in section 4.3.2 , problems due to reference architecture can be measured by scoring or metrics of the implementation. Scoring method is superior in universal use, but there is a problem on accuracy. In using scoring method,

using scenario is effective to make the problems clear. Although metrics got from implementation is numerically accurate, it contains the effect of discrepancy between implementation and the reference architecture. When the tendency indicated by metrics is remarkable enough and the implementation roughly agrees with the reference architecture, we would say the effect of discrepancy is small enough for measuring the magnitude of reference architecture. In this situation we take the method of using metrics preferentially, otherwise we use scoring method.

As for selecting metrics, we find the target of measurement in following viewpoint. Here target means the number of elements or the number of relations, depending on the subject.

- Number of target for measurement
- Number of relationship between elements

For example, problem factor "discrepancy in dependency" in Table 6 is based on dependency. Because dependency is relationship between subsystems, we look for measurement subject at a viewpoint of "Number of relationship between elements".

For reverse engineering of dependency among subsystems, using the DSM (dependency structure matrix) [43] is one of the intelligible methods. DSM is originally invented as a tool for analyzing process of work. Nowadays DSM is also applied to analyze software structure, especially analyzing dependencies between components or subsystems. Several software tools are available.

### (2) Normalizing magnitude of problem factor

In normalizing magnitude of problem factor, it is necessary to prepare classification list such as Table 6 before starting normalization. If the measuring magnitude is done by scoring in five-degree, the result can be used directly as the five-degree of evaluation result. However, if the magnitude of problem is measured by some metrics with numerical output, we have to decide the borders between the five-grades. In the classification list, values at borders of five-grade are essential. Values of border are the concrete values of L1-L4, N1-N4, etc in Table 6. There are two basic approaches to decide those values of border in the classification list.

#### Relative method

This is a method to decide the borders based on the measured values. Once we focus on a certain metrics, measure the same metrics in other portion, in order to survey the relative position of the metrics value of the problematic portion. For example, when we focus on "LOC in subsystem" as metrics for the problem factor of "Broad responsibility in subsystem" in Table 6, we take LOC of other subsystems as well. By discovering the dynamic range of the metric in the system, we decide the values of

borders so that the values in the system can be sorted in five-degrees appropriately. Pros and cons of this method are as follows:

Pros:

The results are sorted appropriately in five-degrees.

Cons:

The results can not be used for judgement based on absolute value.

Absolute method

This is a method to decide the borders based on the past experience of project. Prior to evaluate each problem factor, we prepare the borders based on the past experience of project. If the metrics value is numeric such as number of dependency, LOC, or cyclomatic complexity, we determine the borders from a viewpoint of how the metrics value contributes to the bad effects on the development. Bad effect means bad symptoms on development cost and the quality of products. If there are objective correlation between metrics values and the degree of bad effects, we can utilize them to decide values of borders. However, for projects that do not have such correlation data, we have to decide them in a convincing method. Following is one of the methods to decide absolute value based on the experience of experts.

1. Evaluating the degree of problem for each subsystem
   Evaluate major subsystems by scoring regarding to the same problem. For example, evaluate the largeness of subsystems by H/M/L scoring from a point of view of easiness of handling by experts. Evaluating can be done for similar project in the past.
2. Measure the metrics of those subsystems
   Measure the related metrics for those subsystems for comparison with scoring result. For example, measure LOC for the largeness of subsystems.
3. Determine the borders
   Determine the borders by comparing the result of metrics and scoring result.

Pros and cons of this method are as follows:

Pros:

The results can be used for judgement based on absolute value.

Cons:

The results may be unevenly distributed near some degree.

### 4.4.5 STEP 5: Portfolio Analysis

In this step, we analyze characteristics of each bad smells using a portfolio chart, in which the magnitudes of reference and implemented architecture are mapped onto X-axis and Y-axis, respectively. Table 7 shows outline of this step.

Table 7 Outline of STEP 5

| Items | Descriptions |
|---|---|
| Objective | Analyze characteristics of bad smells |
| Input | List of Bad smells<br>Problem factors with magnitude |
| Output | Portfolio chart |
| Procedure | 1. Plot smells by normalized magnitude of problem factors on PFP plane.<br>2. Roughly prioritize the smells to be fixed. |
| Note | Use candlestick chart for multiple problem factor |

#### (1) Plot Bad Smells

For analyzing bad smells, we have found that it is useful to type bad smells according to the combination of the magnitudes of the problems of reference and implemented architecture. In which the magnitudes of reference and implemented architecture are mapped onto X-axis and Y-axis, respectively. We named this categorization "Problem Factor Portfolio" (PFP) and have outlined it in Figure 8.

We plot each smells by normalized magnitude of problem factors on PFP plane, then analyze them based on absolute and relative position on the PFP plane, in order to grasp the characteristics of problems. Magnitudes in PFP plane are normalized values that we quantified in STEP4. Relative positions of the plot of each smell can be a reference for decision taking of priority of architecture refactoring. Way of plotting varies according to the multiplicity of problem factors.

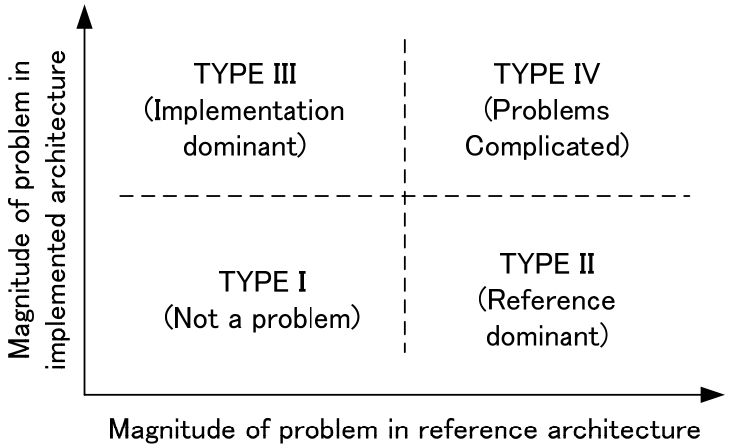


Figure 8 Problem Factor Portfolio (PFP) plane

Single problem factor for a type of problem

If problem factor is single for a type of problem, we plot it simply on the PFP plane. Type of problem means that the problem is based on which reference architecture or implemented architecture. Figure 9 shows an example of plot that both reference and implemented architecture have single problem factor. In Figure 9, a bad smell is illustrated as black circle at the position of 4 at reference architecture and 5 at implemented architecture in magnitude.

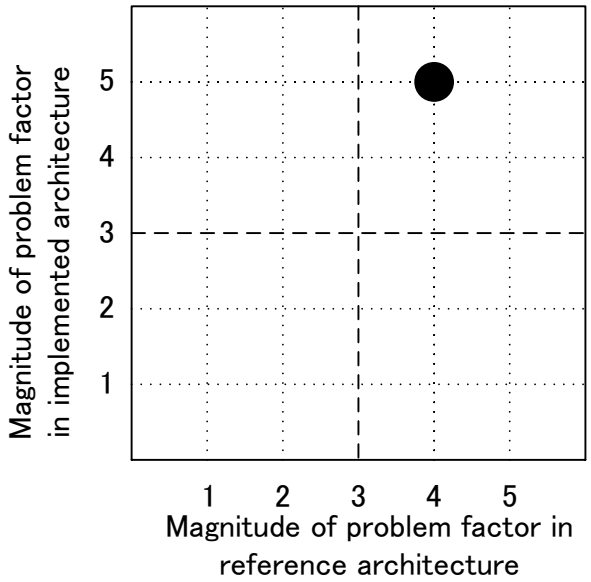


Figure 9 Plot with single problem factor

Multiple problem factors for a type of problem

If problem factors are multiple for a type of problem, we plot the average value on the PFP plane, and plot maximum and minimum values as bars like candlestick chart. Figure 10 shows an example of plot that both reference and implemented architecture have multiple problem factors. In Figure 10, average of bad smell is illustrated as black circle at the position of 3.5 and 4 at reference and implemented architecture. The end of bars represents the minimum and maximum value of the problem factors.

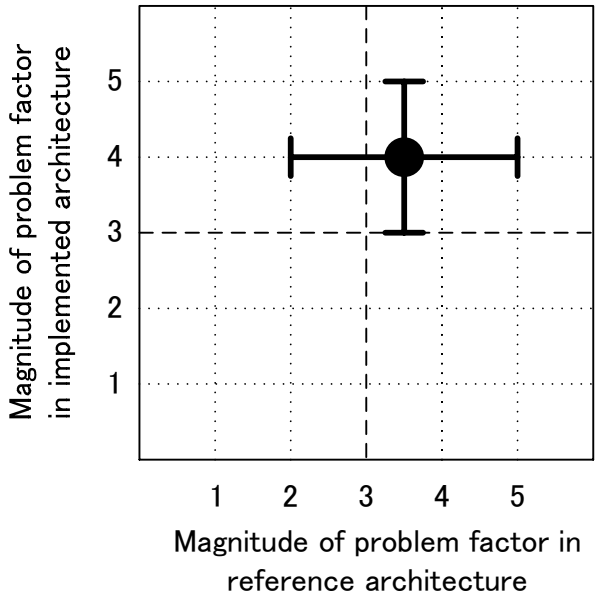


Figure 10 Plot with multiple problem factors

Part of the candlestick chart has following meaning

- Center spot

  Center spot represents the average of problems related to the smell. We can understand the rough tendency of problem factors by the position of center spot.

- Worst value of candlestick

  Center spot represents the average of problems related to the smell. This is used to compare relative values.

- Best value of candlestick

  This is Center spot represents the average of problems related to the smell. This is used to compare relative values.

### (2) Four Typical Groups

Similar to the practice of "Product Portfolio Management" that is used in marketing, in this categorization, the bad smells can be characterized in a four-quadrant matrix. Each type in the four-quadrant matrix represents the following different characteristics. While taking a decision related to the refactoring items, besides referring to each characteristic, we can sort the refactoring items according to the position on the PFP, depending on the situation of the project.

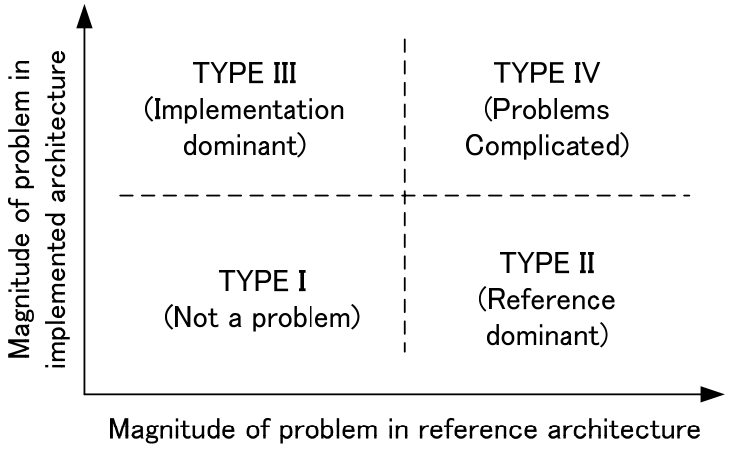


Figure 8 Problem Factor Portfolio (PFP) plane

**TYPE I**—Not a problem

The reference and implemented architecture both have fewer problems as compared to the bad smells in another category. We can judge the priority enforcing refactoring to be low in this case.

**TYPE II**—Problems of reference architecture dominant

This is the case in which the reference architecture is more problematic. This situation typically occurs when the existing reference architecture needs to be modified to fulfill new requirements, which means that it is no longer capable of dealing with upcoming requirements. In short, the current reference architecture has become outdated. Because problem of implemented architecture is low, discrepancy between implementation and reference architecture can not be so big. This situation is caused by such as following reasons:

- External requirement such as industrial standards was revised.
- For smell of TYPE IV, problem of implementation was fixed earlier before dealing with reference architecture.

**TYPE III**－Problems of implemented architecture dominant:

This is the case in which the implemented architecture is more problematic. Even when the reference architecture shows a problem of small magnitudes, it is capable of dealing with upcoming requirements. In some cases in this group, discrepancies are found between the implemented and the reference architecture. In the case of such discrepancies, we recommend that the implementation be refactored early because the problem is concentrated in the implemented architecture. This situation is caused by such as following reasons:

- An implementation skill did not meet the concept or intention of reference architecture.
- Implementation deteriorated over product development.
- Discrepancy between implementation and reference architecture emerged by reactive approach of creating product-line infrastructure.

**TYPE IV**－Problems complicated

The implemented and the reference architecture are both problematic. In this type, even if discrepancies between the implemented and reference architecture are found, it is not recommended that the implementation be refactored in order to recover its consistency with the existing reference architecture. If we refactor the implementation so as to recover the architecture gap without dealing with reference architecture, problems of the reference architecture remain. After all, we need to refactor the implementation again after the revision of the reference architecture. This brings costing twice the labor for a project clearly. So the reference architecture should be refactored, and then the refactoring of the implementation should be carried out to recover the consistency between the implemented and the reference architecture. This situation is caused by such as following reasons:

- Implementation deteriorated under the reference architecture is inappropriate.

### (3) Transition in Four Types

By the progress of refactoring, positions of bad smell changes on problem factor portfolio plane. Generally the magnitude of problem gets smaller by the progress of refactoring. However, depending on the smells, there are transitions that include deterioration in others of simple improvement. We explain the transitions that we think of, by grouping them into improvement, deterioration, and mixed pattern.

1. Group of Improvement

This is a group that improvement is observed in transition. There are typically five transition pattern in this group. Figure 11 shows typical improvement transition pattern. We describe the transition patterns in 1-(a) through 1-(e).

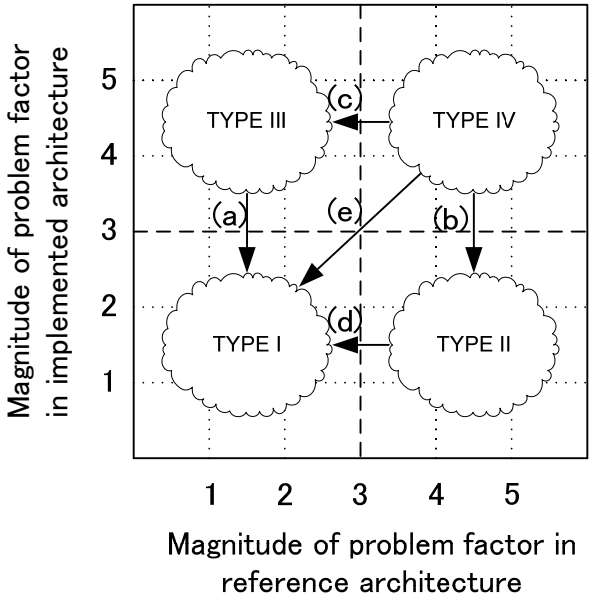


Figure 11 Typical improvement transition patterns

Pattern 1-(a)

This is a pattern of transition from TYPE III to TYPE I. Figure 12 shows the transition on the PFP plane. This transition is seen mostly the implementation is improved when the reference architecture is not so problematic.

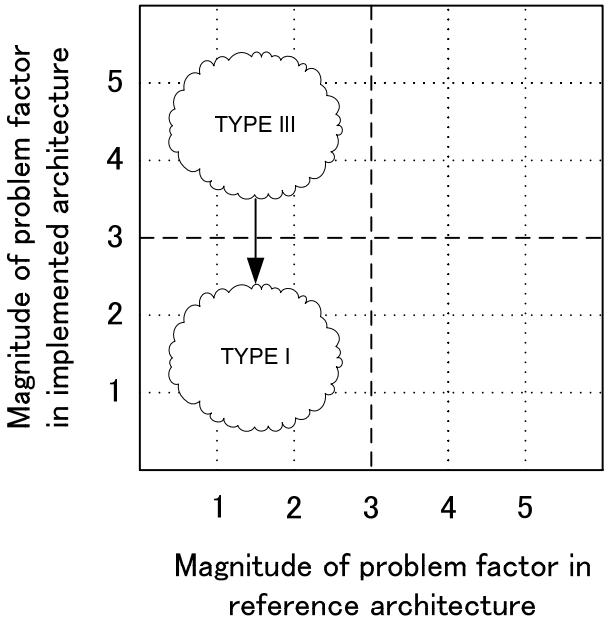


Figure 12 Transition from TYPE III to TYPE I

This pattern can be typically observed in following causes:

- Discrepancies between reference and implemented architecture are dissolved
- Implementation technology is improved

Pattern 1-(b)

This is a pattern of transition from TYPE IV to TYPE II. Figure 13 shows the transition on the PFP plane. This transition is seen mostly the implementation is improved when the problem of reference architecture are left.

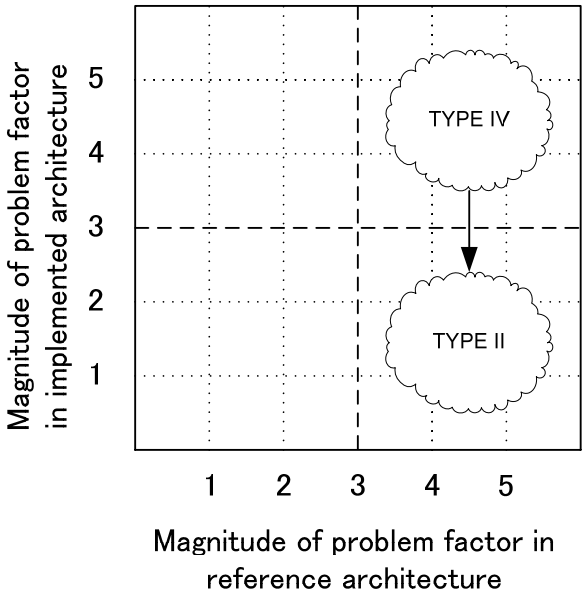


Figure 13 Transition from TYPE IV to TYPE II

This pattern can be typically observed in following causes:

- Problems on implementation are fixed without fixing the problems on reference architecture.
- Implementation technology is improved

Pattern 1-(c)

This is a pattern of transition from TYPE IV to TYPE III. Figure 14 shows the transition on the PFP plane. This transition is seen mostly the reference architecture is improved under the problems on the implemented architecture is left.

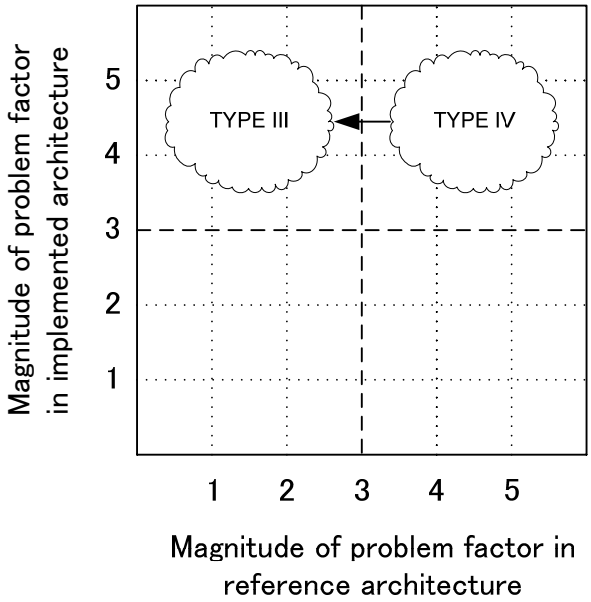


Figure 14 Transition from TYPE IV to TYPE III

This pattern can be typically observed in following causes:

- The reference architecture is improved prior to change the implementation.
- On the way of overall architecture refactoring from TYPE IV to TYPE I.

Pattern 1-(d)

This is a pattern of transition from TYPE II to TYPE I. Figure 15 shows the transition on the PFP plane. This transition is seen when the reference architecture is improved under the condition that the implementation is not so problematic.

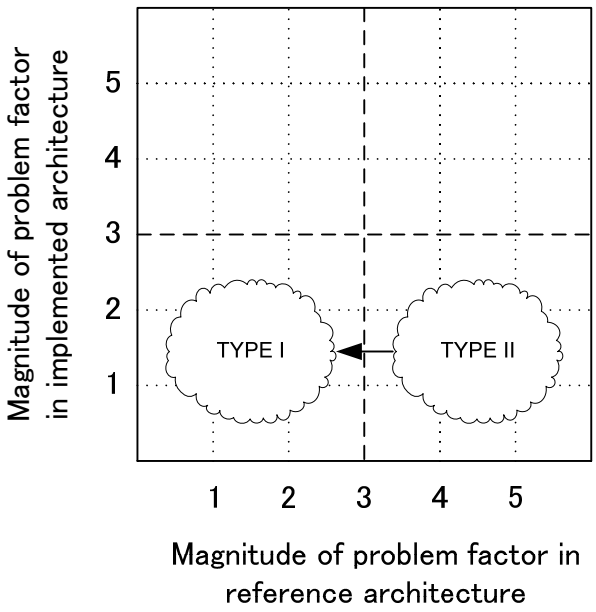


Figure 15 Transition from TYPE II to TYPE I

This pattern can be typically observed in following causes:

- The reference architecture is improved and the related refactoring on the implementation followed immediately.

Pattern 1-(e)

This is a pattern of transition from TYPE IV to TYPE I. Figure 16 shows the transition on the PFP plane. This transition is seen when the reference and implemented architecture is fixed at a time.

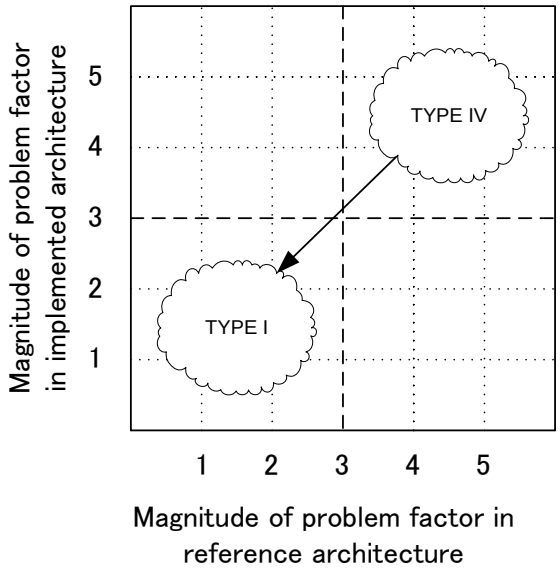


Figure 16 Transition from TYPE IV to TYPE I

This pattern can is typically observed in following causes:

- The reference architecture is improved and the related refactoring on the implementation followed immediately.

2. Group of Deteriorating

This is a group that deterioration is observed in transition. There are typically five transition pattern in this group. Figure 17 shows typical improvement transition pattern. We describe the transition patterns in 2-(a) through 2-(e).

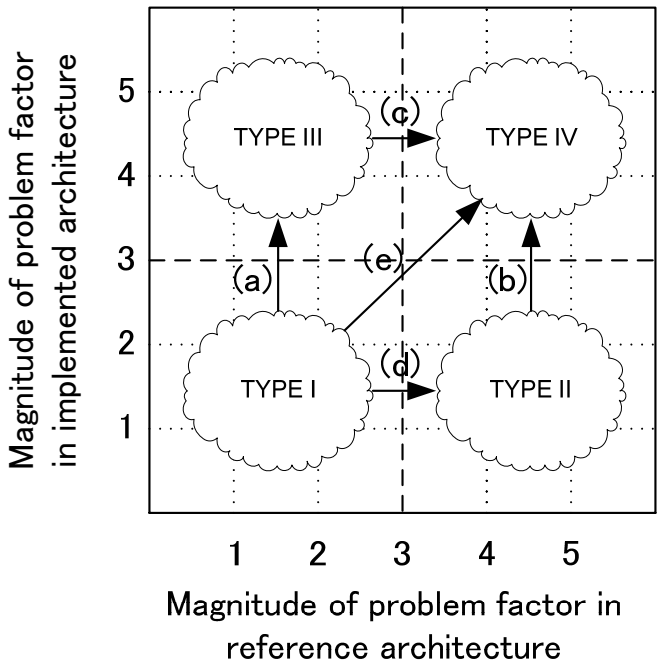


Figure 17 Typical deteriorating transition patterns

These deteriorations are seen under following situations:

- Along with the progress of project
- On the way of overall refactoring activities

Pattern 2-(a),2-(b)

This is a pattern of transition from TYPE I to TYPE III, and TYPE II to TYPE IV. Figure 18 shows the transitions on the PFP plane. These transitions are seen when the implementation is deteriorated regardless of the magnitude of problem factor of the reference architecture.

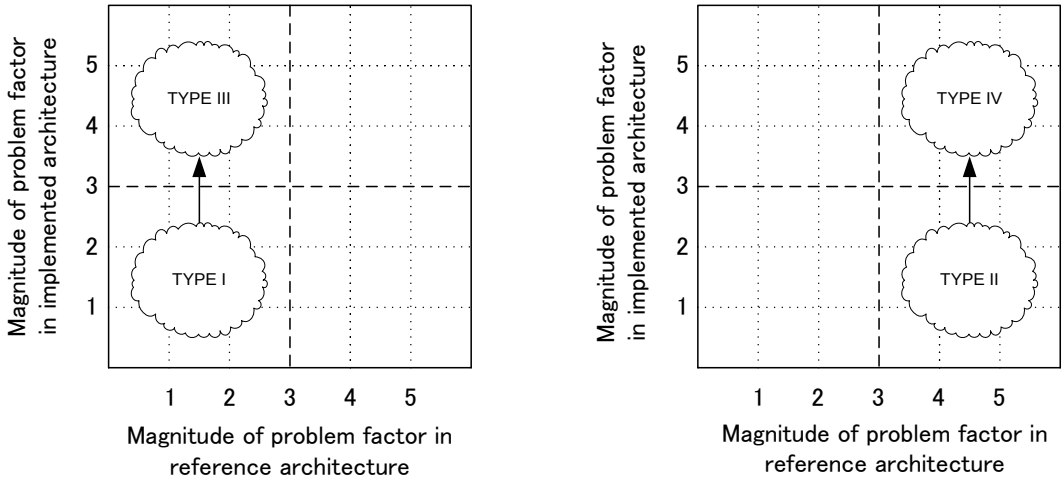


Figure 18 Transition from TYPE I to TYPE III, TYPE II to TYPE IV

Such deteriorations on implementation typically occur on following causes:

- Changes on the implementation are made without understanding the design intention of reference architecture
- Problems on implementation technology

Pattern 2-(c),2-(d)

This is a pattern of transition from TYPE III to TYPE IV, and TYPE I to TYPE II. Figure 19 shows the transitions on the PFP plane.

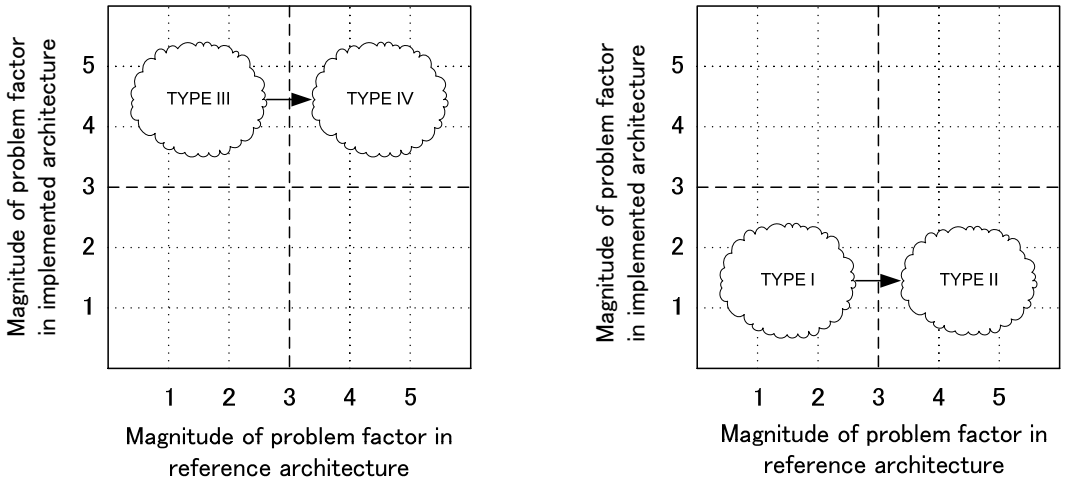


Figure 19 Transition from TYPE I to TYPE II, TYPE III to TYPE IV

Such increase of magnitude of problem factor on the reference architecture typically occurs on following causes:

- These transitions are seen when the problem of the reference architecture emerged because of the change of outside requirements. Scenario based evaluation is good at eliciting these kind of problems, rather than using metrics.

Pattern 2-(e)

This is a pattern of transition from TYPE I to TYPE IV.. Figure 20 shows the transitions on the PFP plane.

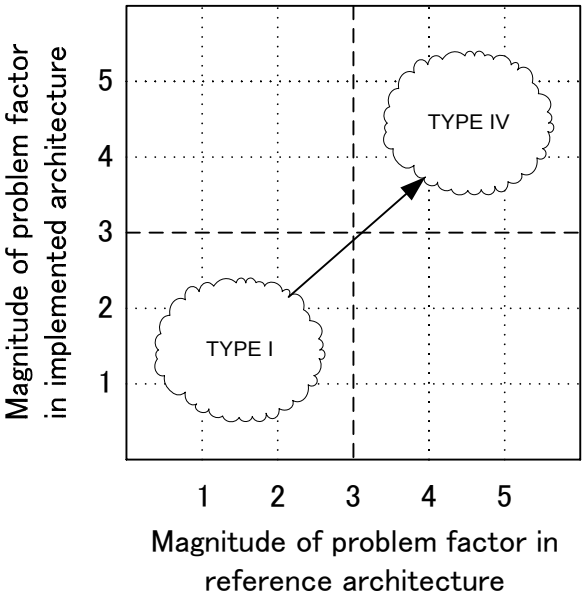


Figure 20 Transition from TYPE I to TYPE IV

Such increase of magnitude of problem factor on both the reference and implemented architecture typically occurs on following causes:

- Combination of other transition.

3. Group of Mixture

This is a group that deterioration and improvement occur at same time. Figure 17 shows typical improvement transition pattern. We describe the transition patterns in 3-(a) an 3-(b).

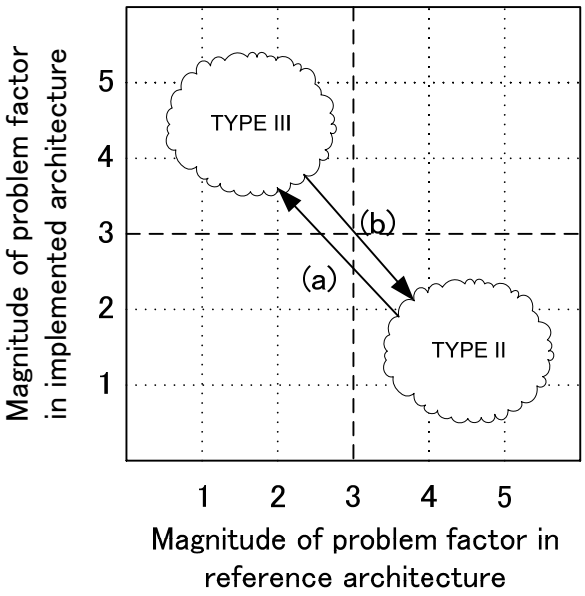


Figure 21 Typical deteriorating transition patterns

Pattern 3-(a)

This is a pattern of transition from TYPE II to TYPE III. Figure 22 shows the transitions on the PFP plane.

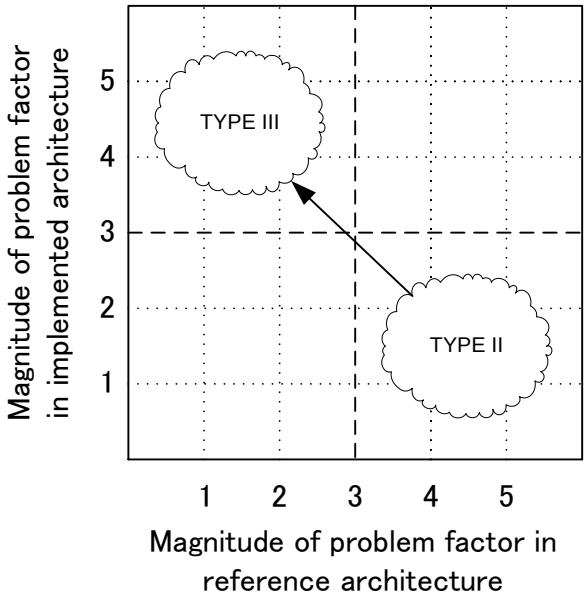


Figure 22 Transition from TYPE II to TYPE III

Such transition typically occurs on following causes:

- Architecture gap appeared because the change of reference architecture at TYPE II.
- On the way of refactoring reference architecture at TYPE II.

Pattern 3-(b)

This is a pattern of transition from TYPE III to TYPE II. Figure 23 shows the transitions on the PFP plane.

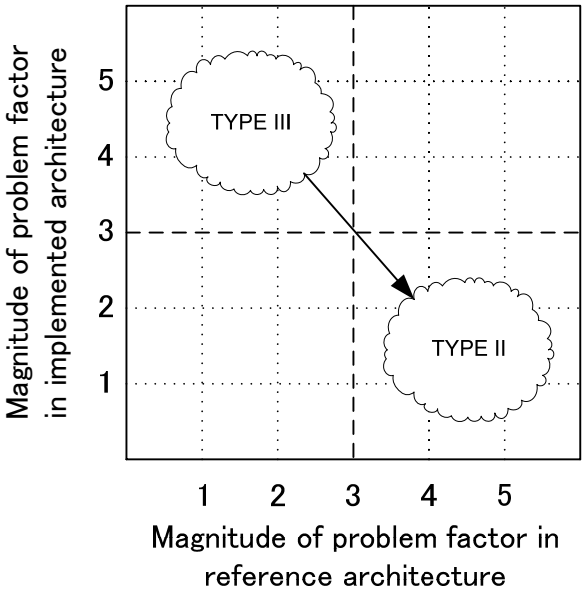


Figure 23 Transition from TYPE III to TYPE II

Such transition typically occurs on following causes:

- Both transition of 1-(a) and 2-(c) occurred.

### 4.4.6 STEP 6: Judge Priority of Refactoring

Considering effort for refactoring calculated in STEP3, Magnitude of each refactoring items in STEP4, Characteristics of bad smells in STEP5, we judge the priority of refactoring items to be performed. Table 8 shows outline of this step.

Table 8 Outline of STEP 6

| Items | Descriptions |
|---|---|
| Objective | Prioritize Refactoring Items |
| Input | Result of problem factor portfolio analysis<br>Effort for refactoring<br>Magnitude of problems<br>Characteristics of problems<br>Situation of the project |
| Output | Prioritized refactoring items |
| Procedure | 1. Roughly categorize smells by portfolio analysis<br>2. Select problems to deal with<br>3. Prioritize refactoring items |
| Note | Prioritize the refactoring items that have good ratio of the magnitude of problem per refactoring effort. |

**(1) Categorizing by portfolio**

Basic ideas for dealing with those smells are as followings;

**TYPE I**－Not a problem

This is a group that problems of both reference and implemented architecture are low. Because the refactoring needs are considered to be low, basically we left these types of bad smells.

**TYPE II**－Problems of reference architecture dominant

This is a group that problems of reference architecture are relatively higher than problems of implemented architecture. Way of dealing with the problems differs depending on the history of the problem. Histories are as followings:

- Change of outside requirement.
  Requirement such as industrial standards was revised. If the changed requirement last in future, we deal with the refactoring of reference

architecture.

- On the way of refactoring from TYPE IV
  It is the case of on the way of refactoring from TYPE IV aiming at TYPE I, problem of implementation was fixed earlier before dealing with reference architecture. We continue refactoring by continuing refactoring of reference architecture.

**TYPE III**－Problems of implemented architecture dominant

This is a group that problems of implemented architecture are relatively higher than problems of reference architecture.
Detailed way of dealing with the problems differs depending on the history of the problem, but basically we deal with the problems by planning refactoring on the implementation. Histories are as followings:

- Problem of implementation technique
  It is the case that the implementation technique did not meet the concept or intention of reference architecture. We consider improving implementation technique to meet the reference architecture.

- Architecture gap over product development
  It is the case that architecture gap between the reference and implemented architecture over products development. We recover the implementation so as to meet the reference architecture.

- Architecture gap by reactive approach
  It is the case that discrepancy between implementation and reference architecture emerged by reactive approach of creating product-line infrastructure. We continue refactoring on the implementation to meet the reference architecture.

**TYPE IV**－Problems complicated

This is a group that problems of both implemented and reference architecture are high. For this group, it is recommended to deal with the problem of reference architecture first. Reason of this is as follows;

If we refactor the implementation first for this type, transition from TYPE IV to TYPE II occurs. In TYPE II, problems of the reference architecture still remain. By improving the reference architecture in TYPE II, new architecture gap between the reference and implemented architecture arise. This causes the transition from TYPE II to TYPE III. For TYPE III, we have to fix the problems of implementation

again to bridge the gaps of architectures. After all, by fixing the problems of implementation first we need three steps to reach TYPE I from TYPE IV, via TYPE II and TYPE III.

Contrary to above, if we choose to fix problems of the reference architecture, we need only two step to reach TYPE I from TYPE IV via TYPE II.

### (2) Select problem factors to deal with

We take bad smells that belongs to the TYPE II, TYPE III, and TYPE IV. For bad smells that belongs to TYPE II, we select problem factors related to the implemented architecture. For bad smells that belong to TYPE III and IV, we select problem factors related to the reference architecture.

### (3) Prioritize refactoring items

For problem factor selected above, we calculate the ratio of the magnitude of problem per refactoring effort. Then we sort the refactoring items by the ratio calculated.

Basic idea for determining priority is as follows;

- Prioritize the refactoring items that have good ratio of the magnitude of problem per refactoring effort.
- Prioritize refactoring items that solves larger problems, if the ratios are the same.
- Select refactoring items within the production costs allowed for refactoring in the project.
- Leave refactoring if big changes may happen in the near future.

### 4.4.7 STEP 7: Execute Refactoring

Execute refactoring according to the judgement in STEP 6. Table 9 shows outline of this step.

Table 9 Outline of STEP 7

| Items | Descriptions |
|---|---|
| Objective | Execute Refactoring for future development efficiency improvement and quality improvement |
| Input | Prioritized refactoring items<br>Left over of refactoring items from previous project |
| Output | Refactored reference architecture<br>Refactored implementation |
| Procedure | Change implementation and related document according to the prioritized refactoring items |
| Note | Select refactoring items to fulfill man-hour restriction in the project.<br>For leftovers, pass them to next project. |

- Refactoring of implementation

  Most of the effort of this step is refactoring of implementation. Figure 24 shows a diagram of the steps involved in refactoring in the course of product development. As can be seen in the diagram, products are released at several intervals and the opportunity to refactor arises between product releases. Architecture assessment involves activities ranging from detecting smells to taking decisions regarding the refactoring items.

  The best time for architecture assessment to take place is immediately following a product release and before the start of development of the next product because we then benefit from the access to smells from both the experience of previous projects and the requirements of the upcoming product.

  In this model, the period of refactoring is shown to be in a relatively early stage of the product development period. This is why refactoring that affects the architecture has a strong impact on development as compared to refactoring in a closed domain.

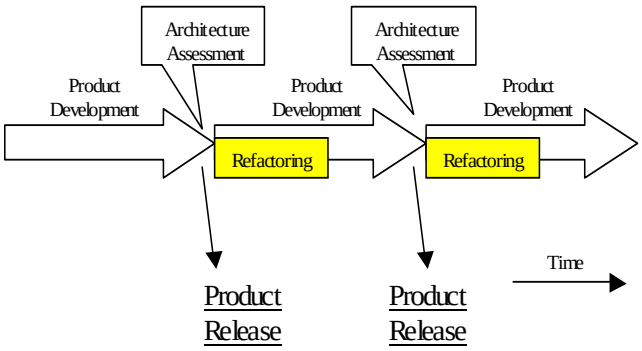


Figure 24 Refactoring Opportunity in Product Development Process

- Refactoring of reference architecture

  Refactoring of architecture in this step means maintenance of architecture documents. It is essential to keep architecture document up to date, in order to not cause confusion on the project. At the same time of maintaining the documents, appropriate announcement of change is also indispensable for the project.

# Chapter 5

# A Sample Case

In this chapter, we outline the system on which we examined our technique retroactively. Reasons of selecting this project for examining our technique are as followings:

- This is a project that actually architecture refactoring is performed.
- Decision takings are done by experts, and the effect of architecture refactoring are confirmed.
- Project data including source code are available for analyzing.

In section 5.1 we describe the projects' overview. In section 5.2 we show the former reference architecture that has been used in development of products. In section 5.3, we describe architecture refactoring that is performed in actual project. Our technique for architecture refactoring is not applied yet at that time.

## 5.1 Domain Characteristics

We explain the characteristics of target project. The target project is development of control software of digital still camera products for consumer. Development scale is about 500,000 lines of codes (LOC) excluding empty lines and comments. Averagely 5 products in 2 series are released in a year. Those two series are concurrently developed, total product release intervals are 2 to 6 month.

Figure 25 shows an image of products load map in a year. Each circle corresponds to product. H1 and H2 belong to the high-end lines, and R1 through R3 belong to the regular lines. H2 and R2 are successors of H1 and R2, R3 is derived model from R1. Both lines belong under the same kind of products, so roughly over 80% of functions are common in both lines. We used common core asset for developing both lines because of abovementioned commonality.

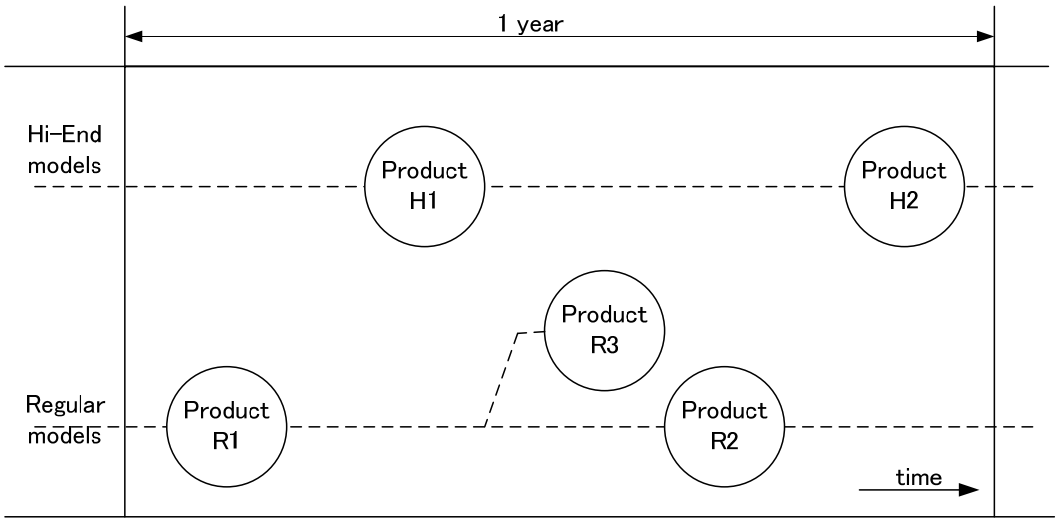


Figure 25 Image of Products Load Map

As for variability, there are two groups. We show extraction of feature model of target product in Figure 26. First one is variability related to hardware key parts such as image sensor or lens unit. This variability is relatively predictable in some extent, because progress of hardware technology is informed beforehand by their suppliers.

Second one is variability related to functions that is mainly actualized by software. Major functions are common among products regardless of high-end line or regular line. There are some exclusive functions for high-end and regular line respectively. These functions grow under development successor models, and some of them are spread into other line. These decisions related to adding functions are often made after starting development, so it is important to have capability to accept such variability that arise later.

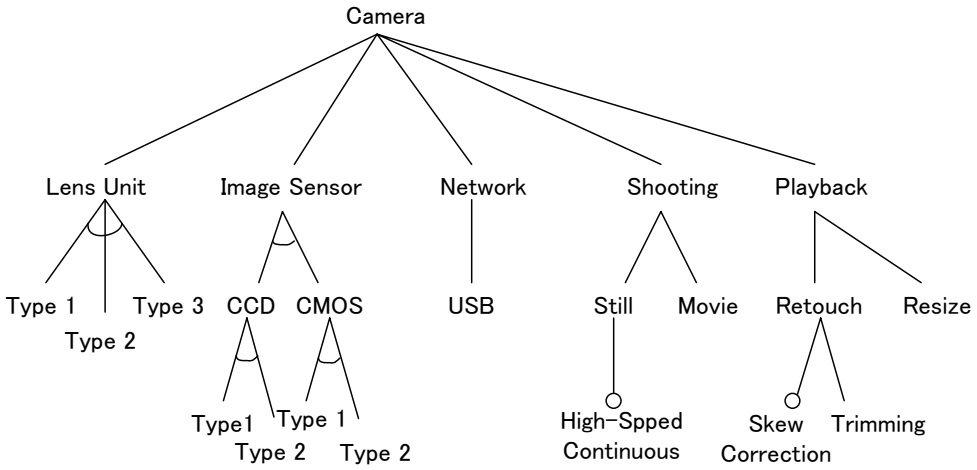


Figure 26 Feature Model of Target Product

In developing multiple similar products, problematic phenomena for efficient development are found out that seems to be caused by the existing software architecture. In order to solve those problems, redesigning from the scratch can be a possible choice. However, as for target project, we thought it is difficult to choose redesigning strategy, considering the intervals of product release and the total scale of the project. Therefore we concluded to improve architecture step-by-step instead of changing at once.

## 5.2 Outline of Reference Architecture

In Figure 27 shows the reference architecture before refactoring. Some adjacent layers are merged to simplify explanation. Totally it is almost layered structure [7], but it does not force strict layering. The reason of allowing non-strict layering is to avoid increase of redundant interpreter in middle layer that merely relay information between upper layer and lover layer. Although we allowed jumped dependency from upper layer to lower layer without relaying at middle layer, we decided to avoid reversed dependency such as from lower layer to middle or upper layer.

In Figure 27, upper layer 'ui' contains highly abstracted procedure such as user interfaces, middle layer 'apl' contains application logic that realize main functions of the products, lower layer 'dev' comprises procedure that treats hardware, and common part 'com' is commonly accessed from all other layers. Although there are several layers in practical system, we omitted those layers that do not related to architecture refactorings mentioned in this paper.

In the reference architecture illustrated in Figure 27, we determined basic access direction as "from upper layer to lower layer". However there exists not negligible number of reversed access on the implementation, and these reversed accesses compose cyclic dependency between layers. Consequently, this was one of the reasons that lower implementation quality especially maintainability.

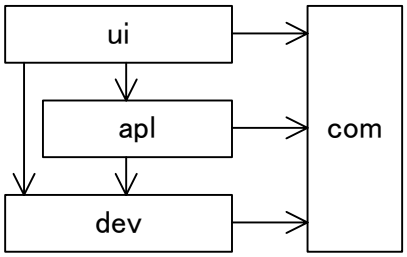


Figure 27 Outline of Reference Architecture

## 5.3 Architecture Refactoring in the Project

In this section, we describe the architecture refactorings that are practically done in the project. In the project we observed following problematic phenomena at that time through the development of product family.

- Phenomenon 1

  Large number of modification effort is needed when adding new function in 'apl' layer. Because the layer takes charge of characterizing each new product on functionalities, modifications always occurs in every product. In those modifications, we found that we needed to understand a lot of related part in the system.

- Phenomenon 2

  Necessary change for adding functions is not closed in specific layer. In other words, there are related portion in other layer that need change together in many cases.

- Phenomenon 3

  Relatively a lot of bugs are detected in a certain layer in testing process of development on every product, compare to other layer.

To solve those problematic phenomena, we decided to change architecture from left side to the right side of Figure 28. On this decision of architecture refactoring, we targeted to reduce cyclic dependency between layers, which is regularly measured on the project.
In practical project, we did not always aware about the place where the problem is (Table 2), which was found via the proposed technique in chapter 4. Although we aware about the causes of the problematic phenomena qualitatively, we did not have ideas of measuring magnitude of problem factors. We refactored architecture based on this awareness.

In the project, we took measures as follows:

- Measures for Phenomenon 1

  We moved portion A from layer 'apl' to 'dev' in order to resolve reversed access from layer 'dev' to 'apl'. Portion A has been located in 'apl' layer, despite it includes direct hardware control.

- Measures for Phenomenon 2

  We introduced layer 'sv' in order to reduce inter-layer dependencies, to push aside parts that related to whole system.

- Measures for Phenomenon 3

  We moved portion B from layer ‘ui’ to layer ‘com’ that contains data commonly referred from other layers.

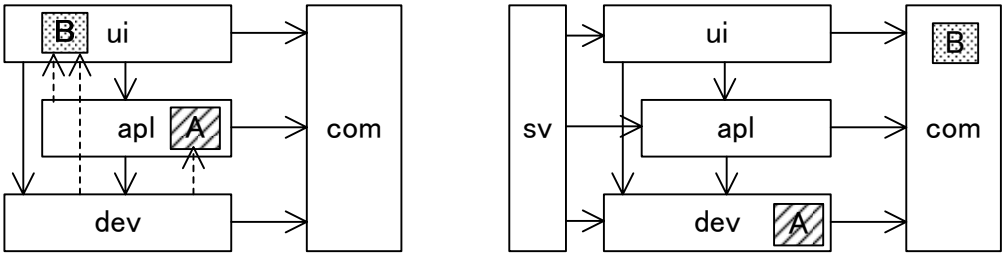


Figure 28 Architecture Refactoring in the Project

## 5.4 Effects by the Architecture Refactoring

By performing architecture refactoring, efficiency of development raised compare to the development under the old architecture. Although we cannot try direct comparison of efficiency such as executing development of the same product on the different architecture, we can say the typical good effect is an increase of the number of products of simultaneous development after the architecture refactoring. This supports the rightness of the judgement by the experts for the refactoring.

# Chapter 6

# Evaluation and Discussion

In this chapter, we describe the evaluation result of proposed technique, using the result of practical architecture refactoring that we introduced in chapter 5. In section 6.1, we explain retroactive application of technique that is used in this evaluation. In section 6.2, we show the result of retroactive application of our proposed technique along with the steps, using actual project data. In section 6.3, summarize the result of application. In section 6.4, we discuss the usefulness of proposed technique by comparing the suggestion that derived by applying technique to the project data and the decision that is made by the experts in the practical project.

## 6.1 Approach of Applying Method

We depict evaluation approach that is used by this research in Figure 29. In the practical project, by investigating problematic phenomena that are observed in the products project, reference architecture and the implementation are refactored. As a result, effects of refactoring are found such as increase of number of products by simultaneous development as we explained in section 5.4 .

In this research, by using same problematic phenomena that are observed in practical project, we tried quantitative analysis of problem factors based on proposed technique. Furthermore, we try to visualize quality attributes of architecture improvement before and after the architecture refactoring.

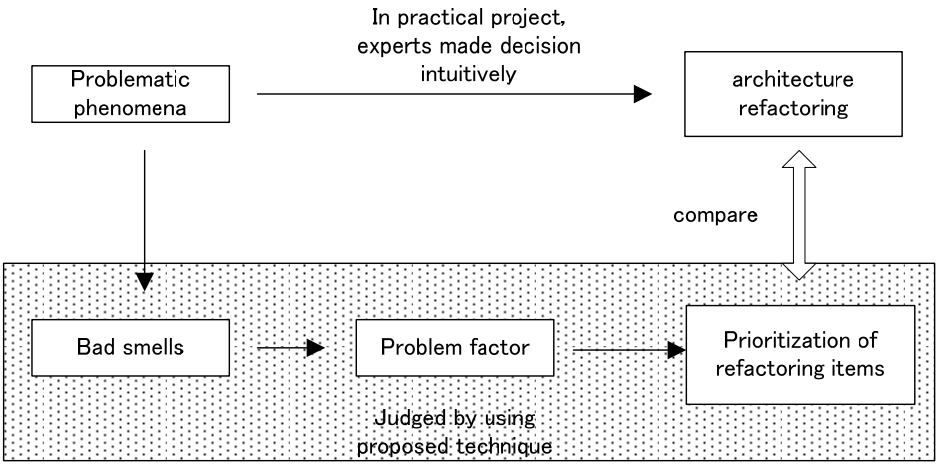


Figure 29 Approach of the Evaluation

## 6.2 Evaluation Using Project Data

In this Section, we describe the result of retroactive application of our proposed technique along with the steps, using actual project data.

### 6.2.1 STEP 1: Select Bad Smells

In this step, we find out bad smells from the experience of development of products. Table 2 shows outline of STEP1.

Table 2 Outline of STEP1

| Items | Descriptions |
|---|---|
| Objective | Sorting out bad smells from project data |
| Input | Bad effects observed in projects of product development |
| Output | List of “Bad smells” |
| Procedure | 1. List up bad effects from the project data<br>2. Pick up bad smells from bad effects |
| Note | Viewpoints for finding bad effects are production cost and quality of the products.<br>Pick up bad effects that emerge in multiple products as bad smells. |

According to the outline of STEP1 shown in Table2, we worked along the step. We used bad effects observed in the project as input, and identified bad smells as output from the viewpoint of emerging in multiple products, as followings:

Input: Bad effects observed in projects of product development

1. Need to check a lot of related part when modifying in layer 'apl' (S1)
2. Need to modify two or more layers for one modification reason (S2)
3. Need big effort on modification on layer 'ui', and easy to cause bugs (S3)
4. Need time to adapt new similar hardware unit (S1)

Output: List of "Bad smells"

Bad smells #1: Need to check a lot of related part when modifying in layer 'apl' (S1)
Bad smells #2: Need to modify two or more layers for one modification reason (S2)
Bad smells #3: Need big effort on modification on layer 'ui', and easy to cause bugs (S3)

### 6.2.2 STEP 2: Find Problem Factors

By analyzing bad smells that is gathered in STEP1, we find out the architecture-related origins that cause those bad smells. Analyzing means investigation into source code and architecture documents. Table 3 shows outline of STEP2.

Table 3 Outline of STEP2

| Items | Descriptions |
|---|---|
| Objective | Finding out problem factors for bad smells |
| Input | Bad smells |
| Output | Problem factors for bad smells |
| Procedure | 1. Seek origins of bad smells<br>2. Exclude origins that are not related to architecture.<br>3. Classify the problem factors into reference-architecture related and implemented-architecture related. |
| Note | Analyze by investigating into source code and architecture documents |

According to the outline of STEP2 shown in Table 3, we worked along the step. We used bad smells that is output of STEP1 as input, and found out problem factors as output, as followings:

Input: List of "Bad smells"

Bad smells #1: Need to check a lot of related part when modifying in layer 'apl' (S1)
Bad smells #2: Need to modify two or more layers for one modification reason (S2)
Bad smells #3: Need big effort on modification on layer 'ui', and easy to cause bugs (S3)

Output: Problem factors

Problem factors for bad smell #1

1. Number of reversed dependency from lower layer (dev) is high, hard to estimate affection scope by modification (Pi3)
2. LOC of layer 'apl' is high because hardware related portion are mixed in (Pr1)

Problem factor for bad smell #2

3. Large number of inter-layer dependency because portion that have global dependency is high (Pr2)

Problem factor for bad smell #3

4. Low maintainability because of cyclomatic complexity value is high (Pi2)
5. Low maintainability because of too many compilation switches (Pi1)
6. Number of reversed dependency from lower layer 'apl' is high, because setting data is located in layer 'ui' (Pr3)
7. LOC of layer 'ui' increases because code that achieve functions is concentrated in layer 'ui'.（Pr1）

### 6.2.3 STEP 3: Plan Refactoring Items

In this step, we plan refactoring items to solve the problem factors got in STEP2. Refactoring item means an individual refactoring plan to resolve the corresponding problem factor. For each refactoring item, we estimate the effort to perform the refactoring actually. Table 4　shows outline of this step.

Table 4 Outline of STEP3

| Items | Descriptions |
|---|---|
| Objective | Planning refactoring items |
| Input | Problem factors |
| Output | Refactoring items to solve problem factors<br>Effort needed for each refactoring items |
| Procedure | 1. List up conceivable refactoring items for each problem factor.<br>2. Estimate man-hour for each refactoring item |
| Note | Viewpoint for planning refactoring items is to decrease magnitude of problems.<br>Estimate effort for refactoring item by using unit man-hour for a refactoring action and total number of refactoring portion. |

According to the outline of STEP3 shown in Table 4, we worked along the step. We used problem factors as input, and planned refactoring items and the estimated man-hour in 3 degrees as outputs, as followings:

Input: Problem factors

1. Number of reversed dependency from lower layer (dev) is high, hard to estimate affection scope by modification (Pi3)
2. LOC of layer 'apl' is high because hardware related portion are mixed in (Pr1)
3. Large number of inter-layer dependency because portion that have global dependency is high (Pr2)
4. Low maintainability because of cyclomatic complexity value is high (Pi2)
5. Low maintainability because of too many compilation switches (Pi1)
6. Number of reversed dependency from lower layer 'apl' is high, because setting data is located in layer 'ui' (Pr3)
7. LOC of layer 'ui' increases because code that achieve functions is concentrated in layer 'ui'（Pr1）

Output1: Refactoring items to solve problem factors

Refactoring Item for problem factor 1

1. Correct reversed dependency at dev→apl (Ri1),

Refactoring Item for problem factor 2

2. Move hardware-related portion into layer ‘dev’ (Rr2)

Refactoring Item for problem factor 3

3. Split the part that have global dependency (Rr3)

Refactoring Item for problem factor 4

4. Lower cyclomatic complexity by splitting long and complex functions (Ri1)

Refactoring Item for problem factor 5

5. Remove extra compilation switches and related source codes (Ri2)

Refactoring Item for problem factor 6

6. Move setting information into commonly accessed layer 'com' (Rr2)

Refactoring Item for problem factor 7

Not investigated

Output2: Effort needed for each refactoring items

1. Correct reversed dependency at dev→apl (Ri1): L
2. Move hardware-related portion into layer ‘dev’ (Rr2): H
3. Split the part that have global dependency (Rr3): M
4. Lower cyclomatic complexity by splitting long and complex functions (Ri1): M
5. Remove extra compilation switches and related source codes (Ri2): L
6. Move setting information into commonly accessed layer 'com' (Rr2): M

### 6.2.4 STEP 4: Quantify Problem Factors

In this step, we quantify and normalize the magnitudes of problem factors identified in STEP2. Table 5 shows outline of this step.

Table 5 Outline of STEP4

| Items | Descriptions |
|---|---|
| Objective | Quantification and normalization of magnitude of problem factors to compare different kinds of problems. |
| Input | Problem factors<br>Classification list for magnitude of problem factors |
| Output | Magnitude for each problem factor in five-grades |
| Procedure | Quantify magnitude of problems using appropriate metrics.<br>Prepare classification list for normalization of magnitude<br>Normalize the magnitude of the problem factor in five-grades by consulting a classification list. |
| Note | Prepare classification list before consulting. |

According to the outline of STEP4 shown in Table 5, we worked along the step. We used problem factors and classification list for magnitude of problem factors as inputs, and classified the result of metrics that represents magnitudes of each problem factor as output as followings:

Input1: Problem factors

1. Number of reversed dependency from lower layer (dev) is high, hard to estimate affection scope by modification (Pi3)
2. LOC of layer 'apl' is high because hardware related portion are mixed in (Pr1)
3. Large number of inter-layer dependency because portion that have global dependency is high (Pr2)
4. Low maintainability because of cyclomatic complexity value is high (Pi2)
5. Low maintainability because of too many compilation switches (Pi1)
6. Number of reversed dependency from lower layer 'apl' is high, because setting data is located in layer 'ui' (Pr3)
7. LOC of layer 'ui' increases because code that achieve functions is concentrated in layer 'ui'（Pr1）

Input2: Classification list for magnitude of problem factors

| Problem factor | Problem Located | Metrics | 5 Degrees of Evaluation Small ← (Magnintude of Problems) → Large | | | | |
|---|---|---|---|---|---|---|---|
| | | | 1 | 2 | 3 | 4 | 5 |
| Broad Responsibility of Subsystems | Reference | LOC of the sybsytem | ＜(L1) | ＜(L2) | ＜(L3) | ＜(L4) | (L4)≦ |
| Dependencies from everywhere | Reference | Number of dependencies | ＜(N1) | ＜(N2) | ＜(N3) | ＜(N4) | (N4)≦ |
| Too many dependencies | Reference | Number of dependencies | ＜(N1) | ＜(N2) | ＜(N3) | ＜(N4) | (N4)≦ |
| Too many compilation switches | Implemented | Dencity of Compilation Switches | ＜(R1)% | ＜(R2)% | ＜(R3)% | ＜(R4)% | (R4)%≦ |
| Long functions | Implemented | Average LOC per functions | ＜(L1) | ＜(L2) | ＜(L3) | ＜(L4) | (L4)≦ |
| High complexity | Implemented | Avarage of Cyclomatic Complexity | ＜(C1) | ＜(C2) | ＜(C3) | ＜(C4) | (C4)≦ |
| High connectivity | Implemented | Number of dependencies | ＜(N1) | ＜(N2) | ＜(N3) | ＜(N4) | (N4)≦ |
| Different components | Implemented | Numbe of different components | ＜(F1) | ＜(F2) | ＜(F3) | ＜(F4) | (F4)≦ |
| Discrepancy in Dependency | Implemented | Number of dependencies | ＜(N1) | ＜(N2) | ＜(N3) | ＜(N4) | (N4)≦ |

Output: Magnitude for each problem factor in five-grades

Magnitude of problem factor 1: 5

Magnitude of problem factor 2: 5

Magnitude of problem factor 3: 4

Magnitude of problem factor 4: 5

Magnitude of problem factor 5: 5

Magnitude of problem factor 6: 4

Magnitude of problem factor 7: 4

### 6.2.5 STEP 5: Portfolio Analysis

In this step, we analyze characteristics of each bad smells using a portfolio chart, in which the magnitudes of reference and implemented architecture are mapped onto X-axis and Y-axis, respectively. Table 7 shows outline of this step.

Table 7 Outline of STEP 5

| Items | Descriptions |
|---|---|
| Objective | Analyze characteristics of bad smells |
| Input | List of bad smells<br>Problem factors with magnitude |
| Output | Portfolio chart |
| Procedure | 1. Plot smells by normalized magnitude of problem factors on PFP plane.<br>2. Prioritize the smells to be fixed. |
| Note | Use candlestick chart for multiple problem factor |

According to the outline of STEP5 shown in Table 7, we worked along the step. We used bad smells and problem factors with magnitude as input, and depicted portfolio chart as output, as followings:

Input1: List of bad smells

Bad smells #1: Need to check a lot of related part when modifying in layer 'apl' (S1)

Bad smells #2: Need to modify two or more layers for one modification reason (S2)

Bad smells #3: Need big effort on modification on layer 'ui', and easy to cause bugs (S3)

Input2: Problem factors with magnitude

Problem factors for bad smell #1:

1. Number of reversed dependency from lower layer (dev) is high, hard to estimate affection scope by modification (Pi3): Magnitude=5
2. LOC of layer 'apl' is high because hardware related portion are mixed in (Pr1): Magnitude=5

Problem factors for bad smell #2:

3. Large number of inter-layer dependency because portion that have global dependency is high (Pr2): Magnitude=4

Problem factors for bad smell #3:

4. Low maintainability because of cyclomatic complexity value is high (Pi2): Magnitude=5

5. Low maintainability because of too many compilation switches (Pi1): Magnitude=5
6. Number of reversed dependency from lower layer 'apl' is high, because setting data is located in layer 'ui' (Pr3): Magnitude=4
7. LOC of layer 'ui' increases because code that achieve functions is concentrated in layer 'ui' (Pr1): Magnitude=4

Output: Portfolio chart

Procedure 1: Plot smells by normalized magnitude of problem factors on PFP plane.

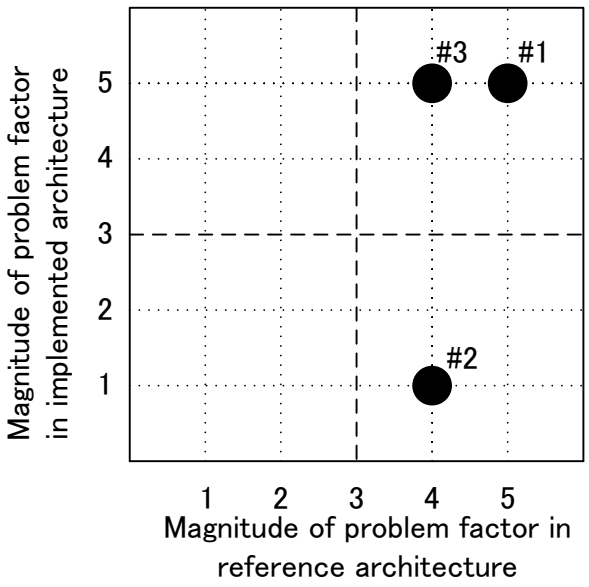


Figure 30 Portfolio chart as an output of STEP5

・For smells that have only problems on reference architecture, we assumed the magnitude of reference architecture to 1 as a most low value.
・For smells that have two or more problem factor in the same type of architecture, we used average of them.

Procedure 2: Prioritize the smells to be fixed.

We identified the order of magnitude of problem are #1,#3,and #2.

### 6.2.6 STEP 6: Judge Priority of Refactoring

Considering effort for refactoring calculated in STEP3, Magnitude of each refactoring items in STEP4, Characteristics of bad smells in STEP5, we judge the priority of refactoring items to be performed. Table 8 shows outline of this step.

Table 8 Outline of STEP 6

| Items | Descriptions |
|---|---|
| Objective | Prioritize Refactoring Items |
| Input | Result of problem factor portfolio analysis<br>Effort for refactoring<br>Magnitude of problems<br>Situation of the project |
| Output | Prioritized refactoring items |
| Procedure | 1. Roughly categorize smells by portfolio analysis<br>2. Select problems to deal with<br>3. Prioritize refactoring items |
| Note | Prioritize the refactoring items that have good ratio of the magnitude of problem per refactoring effort. |

According to the outline of STEP6 shown in Table 8, we worked along the step. We prioritized refactoring items by following steps using output of abovementioned steps as inputs of this step.

Procedure 1: Roughly categorize smells by portfolio analysis

We took bad smells that belong to TYPE II, TYPE III, and TYPE IV. In this case, all bad smells fit to the condition.

Bad smells #1: Need to check a lot of related part when modifying in layer 'apl' (S1)
Bad smells #2: Need to modify two or more layers for one modification reason (S2)
Bad smells #3: Need big effort on modification on layer 'ui', and easy to cause bugs (S3)

Procedure 2: Selected problem factors to deal with:

Problem factors for bad smell #1:

2. LOC of layer 'apl' is high because hardware related portion are mixed in (Pr1): Magnitude=5

Problem factors for bad smell #2:

3. Large number of inter-layer dependency because portion that have global

dependency is high (Pr2): Magnitude=4

Problem factors for bad smell #3:

6. Number of reversed dependency from lower layer 'apl' is high, because setting data is located in layer 'ui' (Pr3): Magnitude=4
7. LOC of layer 'ui' increases because code that achieve functions is concentrated in layer 'ui' (Pr1): Magnitude=4

Procedure 3: Prioritize refactoring items

2. Move hardware-related portion into layer ‘dev’ (Rr2):
   Effort=H, Magnitude=5, Ratio of Magnitude/Effort =1.67
3. Split the part that have global dependency (Rr3):
   Effort=M, Magnitude=4, Ratio of Magnitude/Effort=2.00
6. Move setting information into commonly accessed layer 'com' (Rr2):
   Effort=M, Magnitude=4, Ratio of Magnitude/Effort= 2.00

For calculation of ration of magnitude/effort, we applied value 3/2/1 for effort of H/M/L.

Output: Prioritized refactoring items

By considering the ration of magnitude of problems and effort for refactoring, we got prioritized refactoring items as followings:

First priority: (Ratio = 2.00)

3. Split the part that have global dependency (Rr3)

Second priority: (Ratio =2.00)

6. Move setting information into commonly accessed layer 'com' (Rr2):

Third priority: (Ratio = 1.67)

2. Move hardware-related portion into layer ‘dev’ (Rr2)

### 6.2.7 STEP 7: Execute Refactoring

Execute refactoring according to the result of judge in STEP 6. Table 9 shows outline of this step.

Table 9 Outline of STEP 7

| Items | Descriptions |
|---|---|
| Objective | Execute Refactoring for future development efficiency improvement and quality improvement |
| Input | Prioritized refactoring items<br>Left over of refactoring items from previous project |
| Output | Refactored reference architecture<br>Refactored implementation |
| Procedure | Change implementation and related document according to the prioritized refactoring items |
| Note | Select refactoring items to fulfill man-hour restriction in the project.<br>For leftovers, pass them to next project. |

We did not examined execution of refactoring in this study, because main objective of this study is to confirm the decision taking result by the proposed technique.

## 6.3 Summary of Applying the Technique

We show the result of analysis applying STEP1 through STEP4 of our technique in Table 10. Bad smells in Table 10 are the same bad smells that referred in architecture migration at the project. Information written in Table 10 is obtained by analyzing source code of those days in the project. All bad smells and some problem factors are known in the project, others are got by latter analysis. Magnitudes of problem factor in Table 10 are normalized values. In normalizing, we applied guideline written in Table 1.

We also show the result of problem factor portfolio analysis in Figure 31. Black and white circles represent the characteristics of the bad smells before and after architecture refactoring, respectively.

Table 10 Bad Smells, Problem Factors, and Refactorin Items, on the Project

| | Bad Smells | Problem Located | Problem Factors | Magnitude of Problem Factor | Magnitude of Problem | | Refactoring Items |
|---|---|---|---|---|---|---|---|
| | | | | | Before Refactoring | After Refactoring | |
| #1 | Need to check a lot of related part when modifying in layer 'apl' (S1) | Implemented | 1. number of reversed dependency from lower layer (dev) is high, hard to estimate affection scope by modification (Pi3) | Reversed dependency at dev→apl | 5 | 2 | 1. correct reversed dependency at dev→apl (Ri1) |
| | | Reference | 2. LOC of layer 'apl' is high because hardware related portion are mixed in.(Pr1) | Total LOC of layer 'apl' | 5 | 4 | 2. move hardware-related portion into layer 'dev'(Rr2) |
| #2 | Need to modify two or more layers for one modification reason (S2) | Reference | 3. Large number of inter-layer dependency because portion that have global dependency is high(Pr2) | inter-layer dependency per LOC | 4 | 3 | 3. Split the part that have global dependency(Rr3) |
| #3 | Need big effort on modification on layer 'ui', easy to cause bugs. (S3) | Implemented | 4. Low maintainability because of cyclomatic complexity value is high(Pi2) | Avarage of cyclomatic complexity | 5 | 4 | 4. Lower cyclomatic complexity by splitting long and complexed functions(Ri1) |
| | | | 5. Low maintainability because of too many compilation switches(Pi1) | Density of compilation switches | 5 | 5 | 5. Remove extra compilation switchies and related source codes(Ri2) |
| | | Reference | 6. numver of reversed dependency from lower layer 'apl' is hith, because setting data is located in layer 'ui' (Pr3) | Reversed dependency at apl→ui | 4 | 2 | 6. move setting information into commonly accessed layer 'com' ((Ri2) |
| | | | 7. LOC of layer 'ui' increases because code that achieve functions is concentrated in layer 'ui'.(Pr1) | Total LOC of layer 'ui' | 4 | 4 | Not investigated at this time |

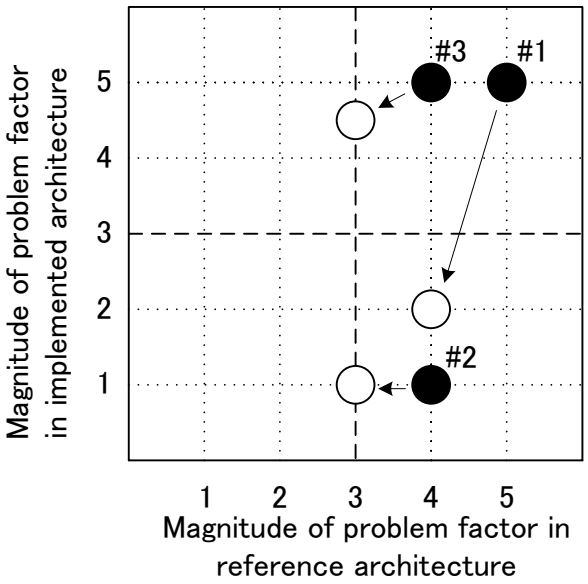


Figure 31 Summary of Portfolio chart

In order to judge the necessity of refactoring, let's see the black circles in the portfolio plot on in Figure 31.

In Figure 31 bad smells #1 and #3 belong to TYPE IV (problems complicated) in Figure 8. It means that we had better to begin refactoring with reference architecture rather than the implemented architecture.

Bad smells #2 belongs to TYPE II (reference dominant) in Figure 8. It means that the necessity of refactoring on reference architecture is indicated, while there are not big problem found in the implemented architecture.

We show the result of individual confirmation result in detail for those bad smells in next section.

## 6.4 Confirmation of Past Instances

We confirmed the past instances that have done in the actual project, by applying the proposed technique. In this section, we show the result of portfolio analysis for each bad smells, that we had intuitive understandings for each bad effect.

Bad smell #1

Figure 32 shows portfolio chart of bad smell #1. Black and white circles represent the characteristics of the bad smells before and after architecture refactoring, respectively. Bad smell #1 belongs to TYPE IV (problems) in Figure 8. where problem of implementation and reference architecture is both high. That is, we had better to begin with refactoring on reference architecture first, because problem factors on both axes are high.

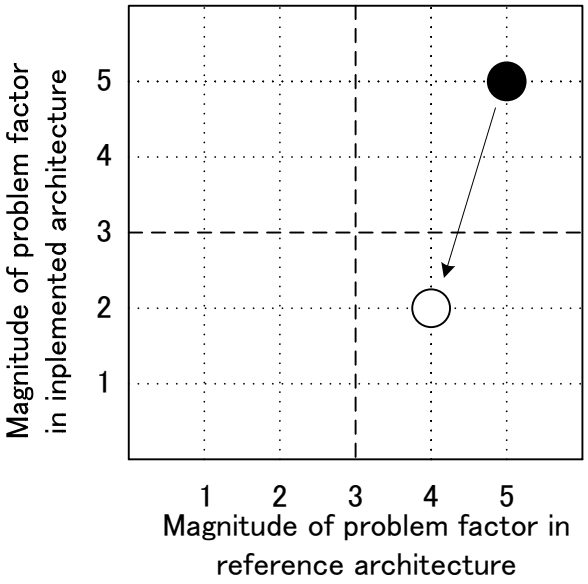


Figure 32 Portfolio Chart of Bad Smell #1

Figure 33 shows the transition of major inter-layer dependency along with produces development. The target system is written in standard-C, dependency count is total of following relations in source code.

- Function calls
- Variable reference
- Macro reference

Prod.1 through Prod.4 corresponds to the products before architecture refactoring, Prod.5 correspond to the product after architecture refactoring. By observing bad smells from Prod.1 to Prod.4, we determined reference architecture refactoring, and performed implemented architecture refactoring between Prod.4 and Prod.5. According to Table 10, metrics of problem factor correspond to the number of reversed dependency between dev-apl. So we discuss further below.

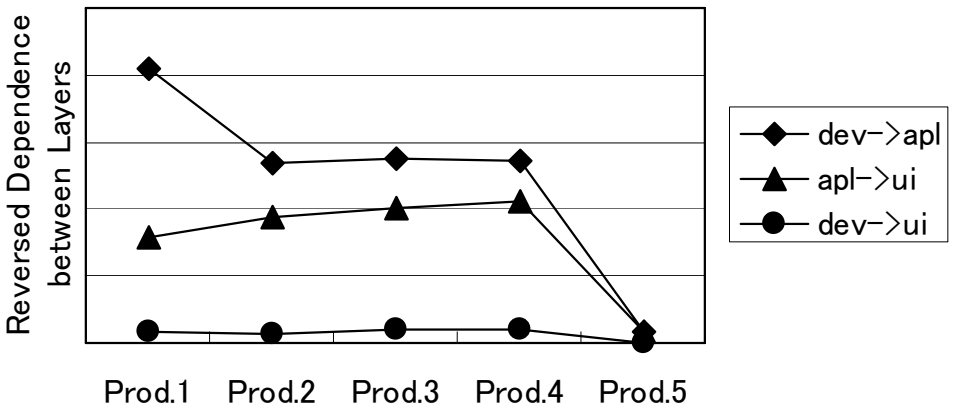


Figure 33 Transition of Reversed Dependency Count

In the project, we aware the problem of reversed dependency, we tried to correct whenever it seems to be necessary. Big decrease of reversed dependency between Prod.1 and Prod.2 is the result of such activity. However, improvements are not found between Prod.2 and Prod.4. This means, almost of easily-correctable reversed dependencies are corrected at Prod.2. In Prod.5, where the architecture refactoring is done, those remaining reversed dependencies are almost corrected. This matches the time of reference architecture refactoring. In other words, to correct those remaining reversed dependencies in the implementation, architecture refactoring of reference architecture was necessary.

Next, we verify the effect of reference architecture refactoring, on the metrics. In Figure 34 show the transition of LOC(Line Of Codes) in major layers. LOC means the number of source code that excludes comments and empty lines. X-axis means time corresponds to development of products, same as Figure 33. The problem factor on reference architecture related to the bad smell #1 is the LOC of layer 'apl'. We can find LOC of each layer decreased at Pord.5 in Figure 34.

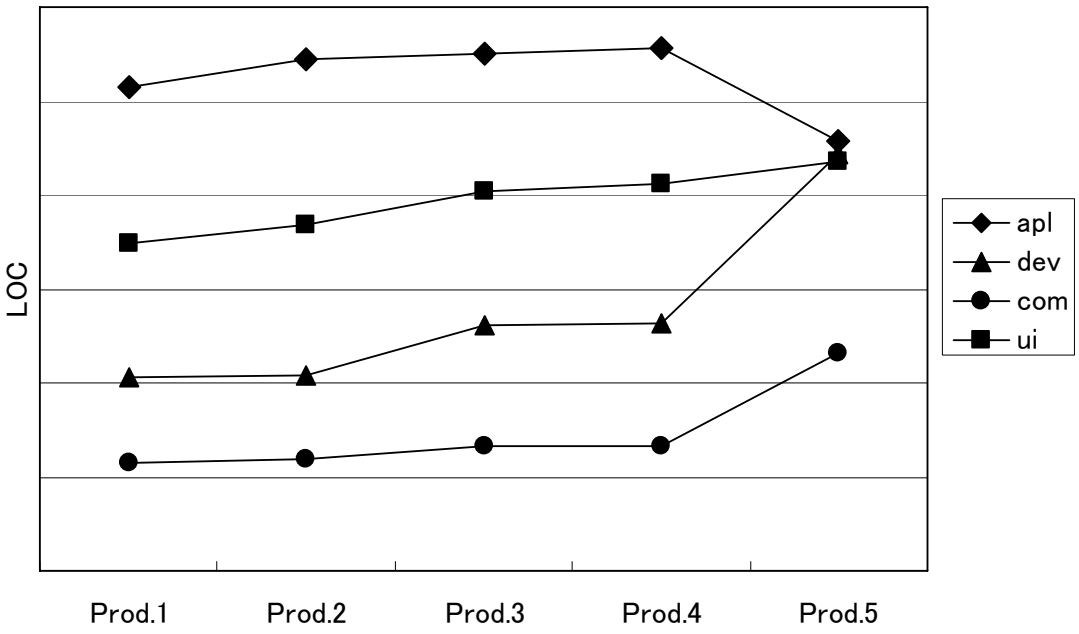


Figure 34 Transition of LOC in Each Layer

In refactoring at Prod.5, we moved hardware-related portion depicted as 'A' in Figure 28. As a result of refactoring, we expected improvement of reversed dependency count from layer 'dev' to layer 'apl', as well as the decrease of LOC of layer 'apl'. In fact, those reversed dependency is almost disappeared in Prod.5, as shown in Figure 33.

That is, for measure to bad smell #1, because the effect of implemented architecture refactoring between Prod.1 and Prod.2 was limited, we need to wait for the reference architecture refactoring at Prod.5. This agrees with the indication of 'refactor reference architecture first' in TYPE IV (problems) in Chapter 4

**Bad smell #2**

Figure 35 shows portfolio chart of bad smell #2. Black and white circles represent the characteristics of the bad smells before and after architecture refactoring, respectively. Bad smell #2 belongs to TYPE II (outdated) in Figure 8, where it is well implemented in accordance with the reference architecture, but there seems to be necessary to refactor reference architecture. We used metrics of dependency ratio per unit LOC between layers for measuring magnitude of reference architecture.

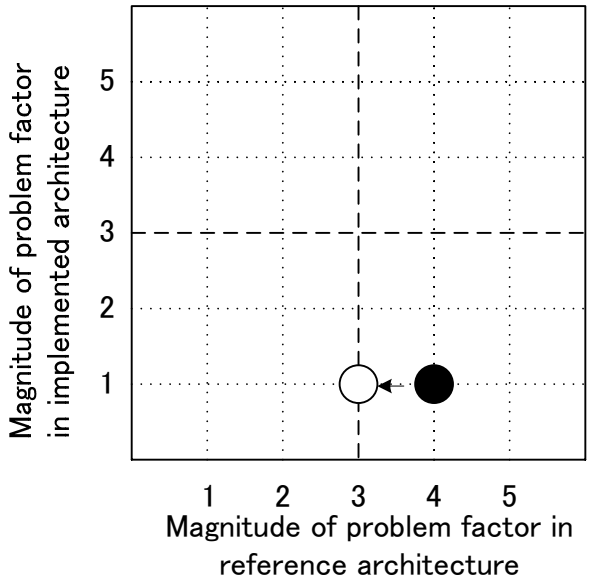


Figure 35 Portfolio Char of Bad Smell #2

Bad smell #2 belongs to TYPE (outdated).in Figure 8, where it is well implemented in accordance with the reference architecture, but there seems to be necessary to refactor reference architecture. We used metrics of dependency ratio per unit LOC between layers for measuring magnitude of reference architecture. Figure 36 shows the transition of dependency ratio per unit LOC.

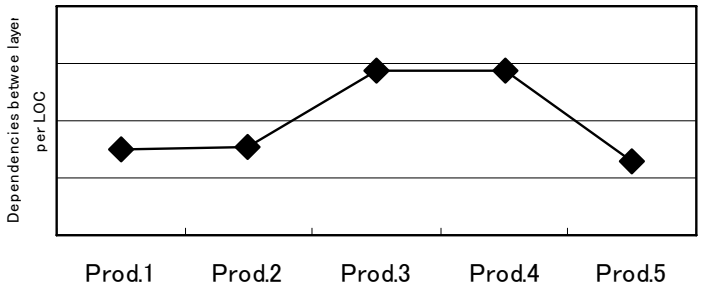


Figure 36 Transition of Inter-Layer Dependency Count per LOC

The number of inter-layer dependencies increased in Prod.3 and Prod.4 because of increase of functions in Prod.3 and Prod.4. However, in Prod.5 where we refactored reference architecture, we found that little improvement compare to Prod.1, despite the number of function did not decreased from Prod.3 and Prod.4.

Through our analysis, refactoring items 3 in Table 10 is derived so as to separate layer 'sv' that have global dependency. The refactoring item 3 is a means to the problem that is caused by scattering of part that have global dependency. It agrees with the decision at real project that aimed to raise independency of layers.

Bad smell #3

Figure 37 shows portfolio chart of bad smell #3. Black and white circles represent the characteristics of the bad smells before and after architecture refactoring, respectively. Bad smell #3 belongs to TYPE IV (problems) in Figure 8, as well as bad smell #1.

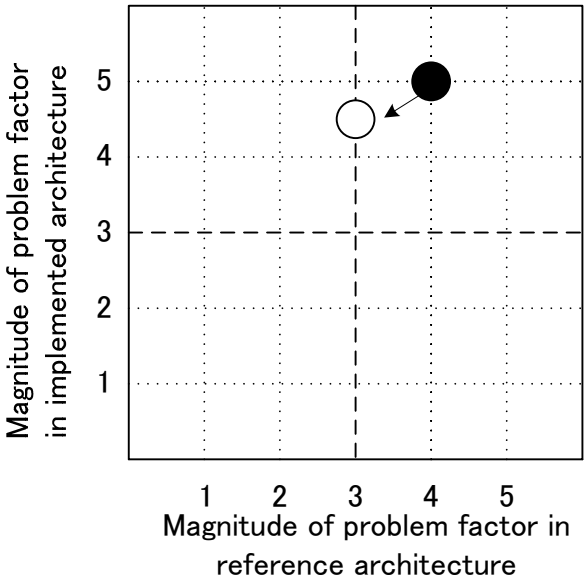


Figure 37 Portfolio Char of Bad Smell #3

We identified problem factors 6 and 7 in Table 10, for problem related to the reference architecture. As for problem factor 6, because almost of all dependency toward layer 'ui' are setting data of the system, it is expected to be corrected by moving setting data from layer 'ui' to layer 'com' . The portion of setting data is depicted as 'B' in Figure 28. This agrees with the result that is done in project.

We chose metrics of reversed dependency from layer 'apl' to layer 'ui' for problem factor 6. As shown in Figure 33, reversed dependency from layer 'apl' to layer 'ui' almost disappeared in Prod.5.

As for problem factor 7, we did not investigate refactoring items at this time. This is because we thought it is inevitable to some extent as long as the layer 'ui' take in charge of functional achievement of the system.

As for problem factor 4 and 5, because those problem factors did not have not necessarily related to the layer structure, it is corrected without waiting for refactoring of reference architecture. We used average complexity to measure problem factor 4. Figure 38 shows the transition of average complexity. Although the average complexity of 'ui' layer is higher than all other layer, the value decreased along with the product development.

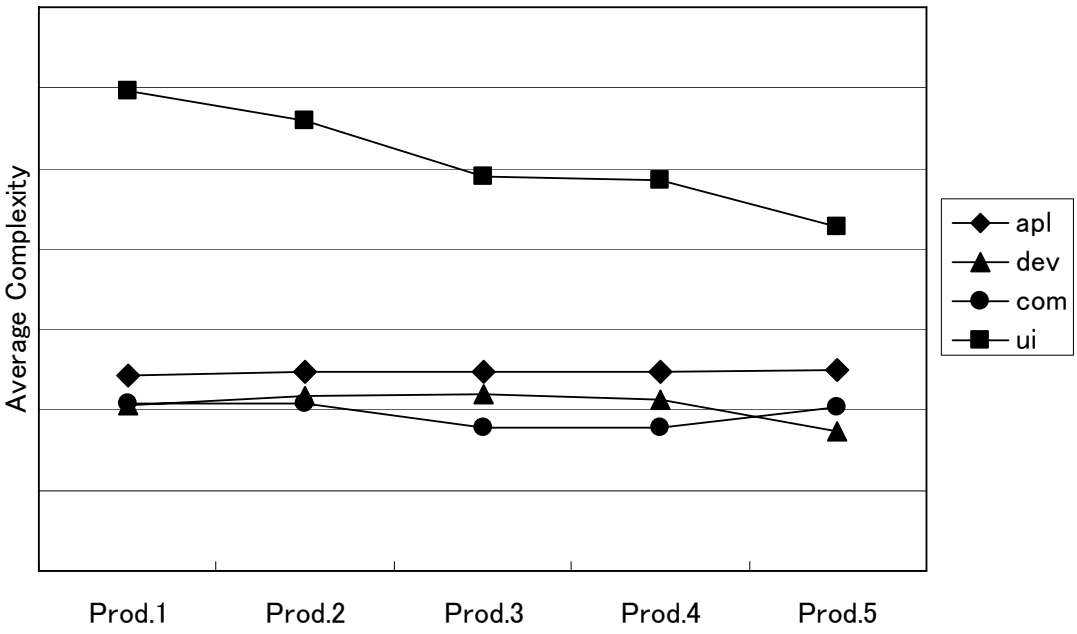


Figure 38 Average Complexity

For problem factor 5, we used metrics of density of compilation switches. Figure 39 shows the transition of density of compilation switches. Compilation switch is one of most easiest in coding and intuitive way for implementing variability. Because of this, there is trend of increasing along with development of products. However the increase of compilation switches introduces bad influence for maintainability of source code. Compilation switch basically controls conditional compilation. So it is similar to if statement, increase of compilation switch is kind of equivalent of increasing cyclomatic complexity. Figure 39 indicate the density of compilation switch of ui layer is high and continuously increasing. In the project, extra compilation switches are removed at all time. As a result, density of apl layer gradually decreased, but density of ui layer increased through development from Prod.1 to Prod.5. This means that layers that always have changes are difficult on decreasing compilation switches against the increase of functions of target product.

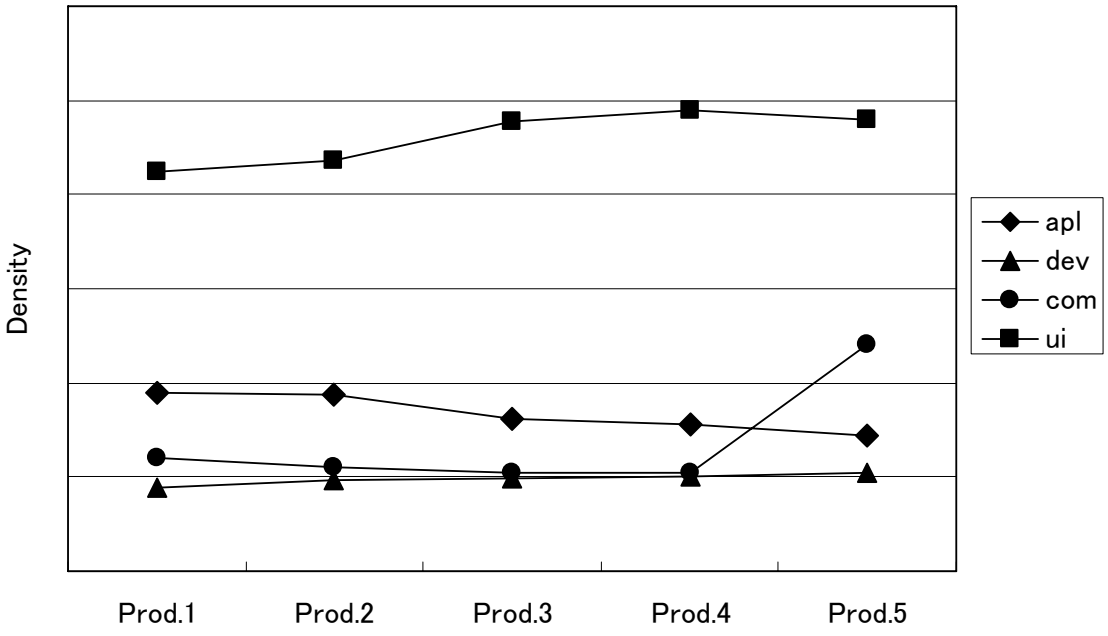


Figure 39 Densities of Compilation Switches

From those result, we can conclude that the indication of TYPE IV “refactor reference architecture first” is not necessarily true, if the problem factor is not strongly related to relationship to other layers.

# Chapter 7

# Related Works

In this chapter, we show difference between this study and related works from the viewpoint of architecture evaluation, migration, and evolution in PLD.

## 7.1 Architecture Evaluation

Regarding to architecture evaluation, various method have been proposed so far, such as ATAM [20], SAAM [21], PuLSE [3], etc. These techniques are effective for evaluation and comparison of architecture. However they do not give direct answers for architecture refactoring that is proceeded along with the development of products.

## 7.2 Architecture Migration

Facts related to refactoring in various software projects are surveyed in [35], in which the refactoring of the architecture is not explicitly distinguished from the refactoring of the source code. P. Bengtsson et al. [4] introduces a method of architecture reengineering that utilizes a specific scenario. Rosso [42] reports on the experience of high-impact refactoring and says that an analysis of architecture violations and "architecture smells" was used to identify refactoring opportunities. They also suggest the possibility of an iterative refactoring process, but this was not mentioned with respect to a method for prioritizing and identifying each refactoring opportunity. They mention the importance of the continuous evolution of software architecture for a family of products. They focus on the assessment of architecture and reports on the basis of their experience using three different assessment methods: scenario-based, performance-based, and experience-based assessments.

## 7.3 Architecture Evolution in PLD

Figure 40 shows process of PuLSE-DSSA that is proposed by Fraunhofer IESE [24], and the general idea of cyclic improvement is based on QIP (Quality Improvement Paradigm) [2] shown in Figure 41. Although, general idea of iterative architecture improvement is shown in those, they need tailoring for adapting to each project, because the description is still abstractive for practical use. Our proposed technique provides a

decision taking method for architecture refactoring in practical software development scene.

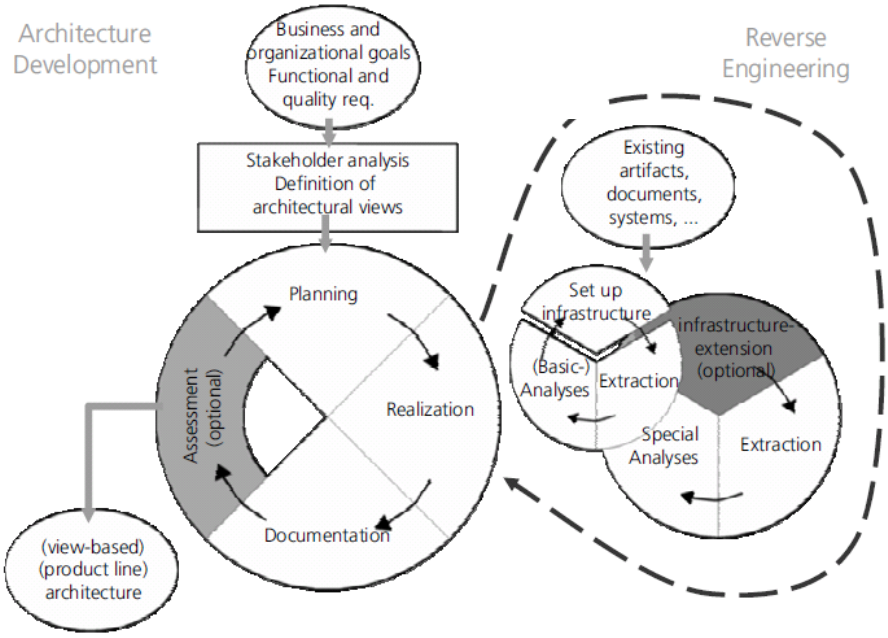


Figure 40 Pulse-DSSA Process

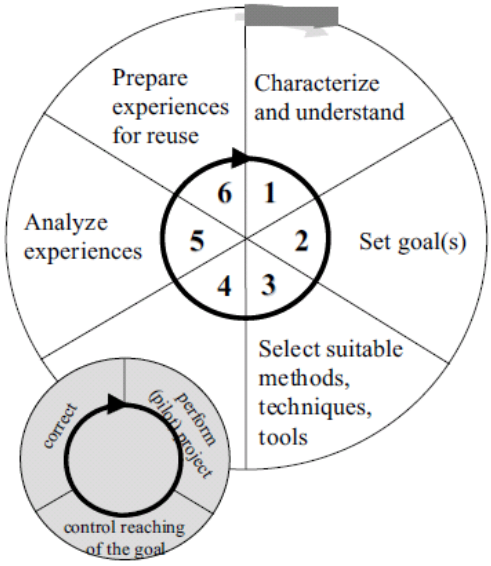


Figure 41 Quality Improvement Paradigm

# Chapter 8

# Conclusion

In this thesis, we proposed technique for decision taking method for architecture refactoring in PLD, and evaluated the method by using actual project data retroactively. Our proposed technique comprises an idea of separating architecture into reference and implemented architecture, a method for comparing magnitude between different kind of problems, and problem factor portfolio as analyzing tools that is useful for decision taking of refactoring.

By verifying our method by using project data, we confirmed that our technique provides similar result to experts about judgment on architecture refactoring. Moreover, we also confirmed the portfolio reflects the trend of problems that occurred in the project.

In a practical scene, it is important to concern the effort that is necessary for architecture refactoring. Our future works includes the refinement of the method so as to consider the balance of effort and effect for the architecture refactoring.

# Bibliography


[1] Atkinson, C., Bayer, J., Bunse, et al.: Component-based Product Line Engineering, Addison-Wesley (2002)

[2] Basili, V. R. and Green, S. Software Process Evolution at the SEL. IEEE Software 11(4):58-66, 1994.

[3] Bayer, J., Flege, O., Knauber, et al.: Pulse: A Methodology to Develop Software Product Lines, *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Reusability* (*SSR '99*), pp.122-131 (1999)

[4] Bengtsson, P. and Bosch, J.: Scenario-Based Software Architecture Reengineering. International Conference on Software Reuse, 1998, 308-317

[5] Boeckle, G., Clements P., McGregor, et al.: Calculating ROI for Software Product Lines, IEEE Software Vol.21, No.3 (2004)

[6] Bourquin, F.: High-Impact Refactoring based on Architecture Violations. *11th European Conference on Software Maintenance and Reengineering*, pp.149-158 (2007)

[7] Buschmann, F., Meunier R., Rohnert. H., et al.: Pattern-Oriented Software Architecture: A System of Patterns, John Wiley & Sons (1996)

[8] Clements, P.: Being Proactive Pays Off, IEEE Software July/August 2002, pp.28-30 (2002)

[9] Clements, P. and Northrop, L.M.: Software Product Lines: Practices and Patterns, Addison-Wesley (2001).

[10] Deelstra, S., Sinnema, M. And Bosch, J.: Product derivation in software product families : a case study, Journal of Systems and Software Vol.74 Issue 2, pp.173-194 (2005)

[11] Duszynski. S., Knodel, J. and Lindvall, M.: SAVE: Software Architecture Visualization and Evaluation. European Conference on Software Maintenance and Reengineering, 2009, 323-324

[12] Fenton, N. E. and Pfleeger, S. L.: Software Metrics: A Rigorous and Practical Approach, PWS Publishing Company (1996)

[13] Fowler, M.: Refactoring: Improving the Design of Existing Code, Addison-Wesley (1999)

[14] Garcia, J. *et al.*: Identifying Architectural Bad Smells. European Conference on Software Maintenance and Reengineering, 2009, 255-258

[15] Gamma, E.,Helm, R.,Johnson, R., and Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley (1995)

[16] Gomaa, H.: Designing Software Product Lines with UML, Addison-Wesley (2005)

[17] http://www.imagix.com/

[18] Jaktman, C. B., Leaney, J., Liu, M., "Structural Analysis of the Software Architecture – A Maintenance Assessment Case Study", in Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1), 1999.

[19] Jilles van Gurp, Bosch, J., Brinkkemper, S. "Design Erosion in Evolving Software Products", position paper, International workshop on the Evolution of Large-scale Industrial Software Applications, ICSM 2003.

[20] Kazman, R., Klein, M., Barbacci, M., et al.: The Architecture Tradeoff Analysis Method, Software Engineering Institute, Technical Report CMU/SEI-98-TR-008 (1998)

[21] Kazman, R., Abowd, G., Bass, L., and Webb, M.: SAAM: A Method for Analyzing the Properties of Software Architectures. *Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, May 1994*, pp.81-90 (1994)

[22] Kerievsky, J.: Refactoring to Patterns, Addison-Wesley (2004)

[23] Kolb, R., John, I., Muthig D., *et al.*: Experiences with Product Line Development of Embedded Systems at Testo AG., *Proc. 10th International Software Product Line Conference*, pp.172-181 (2006)

[24] Kolb, R., Muthig, D., et al.: A case study in refactoring a legacy component for reuse in a product line, Proc. of the 21st IEEE International Conference on Software Maintenance, 2005. ICSM'05. pp.369-378 (2005)

[25] Krueger, C.: Easing the Transition to Software Mass Customization, *Proceedings of the 4th International Workshop on Software Product Family Engineering*, pp.282-293, Springer (2001)

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

[27] http://www.lattix.com

[28] Linden, F.V.D.: Software Product Families in Europe: The Esaps & Café Projects, IEEE Software, Vol.19, No.4 (2002)

[29] Maki, T., Kishi, T.: Problem Factor Portfolio Analysis for Product Line Architecture Refactoring, *17th Asia-Pacific Software Engineering Conference* (APSEC 2010): Industry Papers (2010)

[30] Maki, T., Kishi, T.: Problem Factor Portfolio Analysis for Product Line Architecture Refactoring, Proceeding of 14th Software Product Line Conference (SPLC 2010) the second volume pp.205-208.

[31] Maki, T., Kishi, T.: A Decision Making Method for Product-Line Architecture Refactoring (submitted to IPSJ Journal) (2013)

[32] Maki, T.: Architecture Migration Using DSM in a Large-Scale Software Project, Proceeding of 14th International DSM Conference (DSMC 2012)

[33] Maki, T.: Architecture Migration in Large-Scale Embedded Software Project, MODULARITY: aosd 13, Special Sessions, SPL Symposium, (AOSD 2013)

[34] Maki, T. and Suzuki, M.: Experiment of Adapting Feature Model for Variability Management in Embedded Systems - Extracting Features from Compilation Switches, SIGSE 2012-SE-175(22) pp.1-8, March, 2012. (In Japanese)

[35] Murphy-Hill, E., Parnin, C., and Black, A. P.: How We Refactor, and How We Know It. In ICSE '09 Proceedings of the 31th International Conference on Software Engineering, 2009, 287-297

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

[37] Parnas, D. L. "Software Aging", in Proceedings of ICSE 1994 pp.280-287, 1994.

[38] Perry, D.E., Wolf, A. L. "Foundations for the Study of Software Architecture", in ACM SIGSOFT Software Engineering Notes, vol 17 no 4, 1992.

[39] Phol, K., Boeckle, G.. and Linden, F.V.D.: Software Product Line Engineering: Foundations, Principles And Techniques, Springer-Verlag New York Inc.(2005)

[40] Pollack, M.: Architecture Refactoring: Improving the Design of Existing Application Architectures, Presentation slides of Microsoft Tech.ed North America (2009)

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

[42] Rosso, C. D.: Continuous Evolution through Software Architecture Evaluation: A Case Study. J. Softw. Maint. Evol.: Res. Pract., 2006, 18, 351-383

[43] Sangal, N: Lightweight Dependency Models for Product Lines. 10th International Software Product Line Conference, pp.228 (2006)

[44] Tischer, C, Schmid, K., et al.: Developing Long-Term Stable Product Line Architectures, *Proc. 16th International Software Product Line Conference*, pp.86-95 (2012)

# Publications

Refereed:

[1] Maki, T., Kishi, T.: Problem Factor Portfolio Analysis for Product Line Architecture Refactoring, *17th Asia-Pacific Software Engineering Conference* (APSEC 2010): Industry Papers (2010)

[2] Maki, T., Kishi, T.: Problem Factor Portfolio Analysis for Product Line Architecture Refactoring, Proceeding of 14th Software Product Line Conference (SPLC 2010) the second volume pp.205-208.

[3] Maki, T., Kishi, T.: A Decision taking Method for Product-Line Architecture Refactoring, IPSJ Journal (conditional acceptance) (2013)

[4] Maki, T.: Architecture Migration Using DSM in a Large-Scale Software Project, Proceeding of 14th International DSM Conference (DSMC 2012)

Non-Refereed:

[5] Maki, T.: Architecture Migration in Large-Scale Embedded Software Project, MODULARITY: aosd 13, Special Sessions, SPL Symposium, (AOSD 2013)

[6] Maki, T. and Suzuki, M.: Experiment of Adapting Feature Model for Variability Management in Embedded Systems - Extracting Features from Compilation Switches, SIGSE 2012-SE-175(22) pp.1-8, March, 2012. (In Japanese)

# Appendix A

# Refactoring

In this chapter, we describe general idea of architecture refactoring. Prior to explain architecture refactoring, we mention source code refactoring in section A.1. In section A.2, we describe extension of refactoring from source code to architecture. In section A.3, we mention scale of architecture refactoring.

## A.1 Source Code Refactoring

Refactoring is well-known technique for changing internal structure of the program without changing its functionality, from the viewpoint of maintainability and portability. Originally this was a technique for maintaining source code.

Fowler covers major techniques of refactoring in [13]. According to Fowler, candidates for refactoring can be detected from several phenomena that are observed on source code, for example "duplicated code", "long method", etc. These are called "bad smells in code" in [13], and 22 kinds of bad smells are introduced there. Basically these bad smells are based on structural observation of source code. Benefit for utilizing these structural characteristics is, even someone who is not so familiar to the system can point out the possibility of problems. However, not all these smelled portions are necessarily contains problems. So necessity of refactoring should be judged individually by considering project's situation.

Regarding source code refactoring in action, Fowler [13] lists refactoring catalog, where 72 kinds of refactoring operation are introduced, for example "extract method", "move method", "remove parameter", and so on. This refactoring catalogue provides guidelines for local rearrangement of source code. By performing these modifications on source code, implementation quality such as maintainability is expected to improve in many cases. Because these refactoring operations are very primitive and elementary, some of these refactoring can be commonly used in architecture refactoring that is mentioned in section A.2.

As upper concept to elemental refactoring, Kerievsky proposes refactoring catalogues in [22], from the viewpoint of design pattern [15]. Compare to Fowler's low level refactoring catalogue, it gives us middle level refactoring goals according to design pattern. Similar to Fowler's approach, Kerievsky also utilizes "code smells" to find

refactoring candidates. They mention 12 kinds of code smells in [22]. Some of the smells are common to Fowler's smells, such as "Duplicated Code", "Long Method", "Lazy Class", and "Large Class".

## A.2 Extension of Refactoring

The general idea of source code refactoring can be extended to software architecture. We call the similar operation to the architecture "architecture refactoring". Architecture refactoring is same as architecture evolution that is done in order to improve quality attributes such as maintainability or portability, with keeping major function of the products that is expected to be developed on the architecture.

Because architecture evolution for adapting change of business and engineering environment can include functional enhancement, it may have different side from source code refactoring that does not change functionality. However, we use the term "architecture refactoring" for such architecture evolution by following reasons:

- There is continuity of products before and after the architecture refactoring
- We don't consider essential change on major functionality
- Basically for improvement of quality attributes for major functionality

## A.3 Scale of Refactoring

Corresponding to design granularity, scale of refactoring can be categorized into source code level and architecture level. Figure 42 shows correspondence between design granularity and refactoring scope. Refactoring scope of source code refactoring is closed within class and package. On the other hand, scope of architecture refactoring covers up to layers. Because the scope is large and it affects is widespread, architecture refactoring should be conducted under appropriate decision.

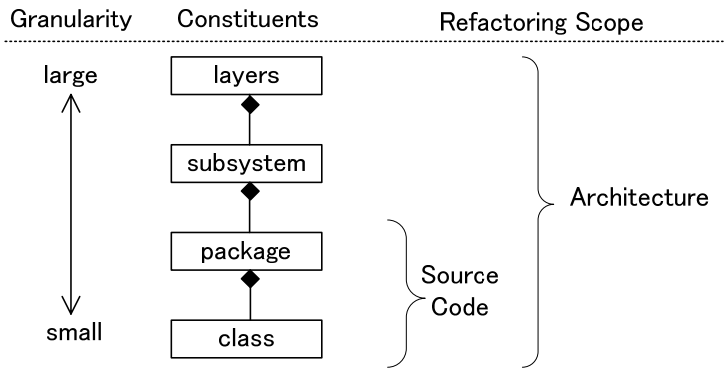


Figure 42 Relationship between Refactoring Scope and Design Granularity

# Appendix B

# Smells and Refactoring Catalogs

## B.1 Definition of Smells

Followings are lists of bad smells that are proposed in previous research. We utilized them as a reference of problem factors on software structure, because they are based on observation of the structure of the implementation.

### B.1.1 Smells by Fowler

- Duplicated Code
- Long Method
- Large Class
- Long Parameter List
- Divergent Change
- Shotgun Surgery
- Feature Envy
- Data Clumps
- Primitive Obsession
- Switch Statements
- Parallel Inheritance Hierarchies
- Lazy Class
- Speculative generality
- Temporary Field
- Message Chains
- Middle Man
- Inappropriate Intimacy
- Alternative Classes with Different Interfaces
- Incomplete Library Class
- Data Class
- Refused Bequest
- Comments

### B.1.2 Code Smells by Kerievsky

- Duplicated Code
- Long Method
- Conditional Complexity
- Primitive Obsession
- Indecent Exposure
- Solution Sprawl
- Alternative Classes with Different Interfaces
- Lazy Class
- Large Class
- Switch Statements
- Combinatorial Explosion
- Oddball Solution

## B.2 Refactoring Catalogs

Followings are refactoring catalogs that are proposed in previous research. Compared with a catalog of Fowler, catalog of Stal includes refactorings more in architecture level that corresponds to refactoring items in our study.

### B.2.1 Refactoring Catalog by Stal

- Rename Entities
- Remove Duplicates
- Introduce Abstraction Hierarchies
- Remove Unnecessary Abstractions
- Substitute Mediation with Adaptation
- Break Dependency Cycles
- Inject Dependencies
- Insert Transparency Layer
- Reduce Dependencies with Facades
- Merge Subsystems
- Split Subsystems
- Enforce Strict Layering
- Move Entities
- Add Strategies
- Enforce Symmetry
- Extract Interface
- Enforce Contract

- Provide Extension Interfaces
- Substitute Inheritance with Delegation
- Provide Interoperability Layers
- Introduce Aspects
- Integrate DSLs
- Add Uniform Support to Runtime Aspects
- Add Configuration Subsystem
- Introduce the Open/Close Principle
- Optimize with Caching
- Replace Singleton
- Separate Synchronous and Asynchronous Processing
- Replace Remote Methods with Messages
- Add Object Manager
- Change Unidirectional Association to Bidirectional

### B.2.2 Refactoring Catalog by Fowler

- Extract Method
- Inline Method
- Inline Temp
- Replace Temp with Query
- Introduce Explaining Variable
- Split Temporary Variable
- Remove Assignments to Parameters
- Replace Method with Method Object
- Substitute Algorithm
- Move Method
- Move Field
- Extract Class
- Inline Class
- Hide Delegate
- Remove Middle Man
- Introduce Foreign Method
- Introduce Local Extension
- Self Encapsulate Field
- Replace Data Value with Object
- Change Value to Reference
- Change Reference to Value
- Replace Array with Object
- Duplicate Observed Data

- Change Unidirectional Association to Bidirectional
- Change Bidirectional Association to Unidirectional
- Replace Magic Number with Symbolic Constant
- Encapsulate Field
- Encapsulate Collection
- Replace Magic Number with Symbolic Constant
- Encapsulate Field
- Encapsulate Collection
- Replace Record with Data Class
- Replace Type Code with Class
- Replace Type Code with Subclasses
- Replace Type Code with State/Strategy
- Replace Subclass with Fields
- Decompose Conditional
- Consolidate Conditional Expression
- Consolidate Duplicate Conditional Fragments
- Remove Control Flag
- Replace Nested Conditional with Guard Clauses
- Replace Conditional with Polymorphism
- Introduce Null Object
- Introduce Assertion
- Rename Method
- Add Parameter
- Remove Parameter
- Separate Query from Modifier
- Parameterize Method
- Replace Parameter with Explicit Methods
- Preserve Whole Object
- Replace Parameter with Method
- Introduce Parameter Object
- Remove Setting Method
- Hide Method
- Replace Constructor with Factory Method
- Encapsulate Downcast
- Replace Error Code with Exception
- Replace Exception with Test
- Pull Up Field
- Pull Up Method
- Pull Up Constructor Body
- Push Down Method

- Push Down Field
- Extract Subclass
- Extract Superclass
- Extract Interface
- Collapse Hierarchy
- Form Template Method
- Replace Inheritance with Delegation
- Replace Delegation with Inheritance
- Tease Apart Inheritance
- Convert Procedural Design to Objects
- Separate Domain from Presentation
- Extract Hierarchy