Studies on Software Architectural Design

by

Tomoji KISHI

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

Supervisor: Professor Takuya Katayama

School of Information Science
Japan Advanced Institute of Science and Technology

June 30, 2002

Abstract

In this paper, we discuss software architectural design methods, especially that in the early phase of software development to find out the design direction for the software.

In architectural design, we examine fundamental software structure considering the requirements on potential software that will be developed on the architecture, in terms of functionalities and quality attributes. Besides, as architecture imposes constraints on following software design, we have to determine the most appropriate design direction, in the early phase, based on information in hand at that time. In this paper, we examine an architectural design method, considering these characteristics.

We make a case study on actual architectural design to clarify that we need to examine the followings in architectural design; the applicability of architectural design alternatives to requirements, relative preferences among applicable candidates, and, in product-line architectural design, the tradeoffs between the appropriateness of architectural candidates to the product-line as a whole and the appropriateness to each member of the product-line. Then we develop the conceptual framework on architectural design, in which we clarify the relationship among various concepts related to software architecture and architectural design.

Based on the above observations, we propose a concrete architectural design method. This method provides the method to analyze requirements utilizing factors that determine quality attributes, separate requirements based on aspect-oriented concepts, categorize requirements for applicability examination, determine preferences using decision-making techniques, and examine tradeoffs for product-line architectural design. We evaluate the techniques based on an actual case of architectural design.

The contributions of the paper are to clarify the conceptual framework of architectural design, and to propose a concrete architectural design method based on it. Furthermore, as the method explicitly handles the criteria, reasons, and the result of design decision, it makes design objective, and helps us to trace the reasoning of the design decision.

Acknowledgments

The authors would like to thank Professor Takuya Katayama for continuous supports and suggestions to the research. We also thank Professor Koichiro Ochimizu, for his comments on the research, especially application of AHP method toward architectural design. Dr. Toshiaki Aoki and members of Katayama laboratory give us helpful comments and support to the research. This research is motivated by ITS on-board system architecture projects. We appreciate every member participated in the projects. We would like to express our gratitude to Ms. Natsuko Noda for intensive discussion on the research.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Software Architectural Design

1.1.1 Background

Software architecture is the fundamental software structure that is composed on the infrastructure of software. Software architectural design is the design of software architecture for target software, and is also quite important for the following reasons;

- Software architecture determines the characteristics of the software based on the architecture. For example, run-time architecture of software is the structure of components provided by the run-time infrastructure, such as threads and resources, and has an impact on run-time quality attributes such as performance and run-time memory size. Development-time architecture of software is the structure of components provided by the development-time infrastructure, such as programming language constructors, and has an impact on development-time quality attributes such as extensibility and reusability.

  Therefore we have to design software architecture so as to satisfy the requirements on quality attributes related to the structure, considering the characteristics of the infrastructure.

- Software architecture has a strong impact on the software development. As software architecture is the fundamental structure, many design decisions depend on the software architecture. If we change the software architecture, that causes serious impact on these design decisions. In this sense, software architectural design affects succeeding software development.

  Therefore, we have to analyze the nature of software, its evolution, product families and development style and design software architecture so as to be steady; otherwise, software development may become quite expensive.

- It is required to make software architectural decisions open. When we want different software to communicate and collaborate with each other, we have to make these software share the same architectural assumptions. For example, software components for Web-computing systems have to share the same architectural assumptions, such as responsibilities of each component, rules of communications, error-handling policies, so as to collaborate correctly.

Therefore, when we develop software, we have to explicitly define the architecture and the assumptions behind that in order to make the architecture open.

1.1.2 Architectural Design

The term ”software architecture” gives the impression that it means important software structure that is the basis of entire structure and governs other software structure. However, it is difficult to distinguish architectural design from ordinary software design by ’importance’, because it is a quite objective notion.

In this paper, we try to clarify software architectural issues by focusing on the requirements on software. Namely, in architectural design, we are to examine not only the requirements on current product but also potential requirements on the product. For example, if we are to design software architecture that will be steady throughout the software evolution, we have to examine potential requirements on the software. Similarly, if we are to develop software spirally, we have to examine development scenarios and examine the requirement on the software in each iteration to make software architecture to be the basis of the development.

In actual software development, we encounter many situations in which we have to consider architectural issues. In our research, we have categorized the situations into the followings:

- Designing architecture for a single product. Here we design software architecture for a product considering the evolution of the software, such as customization and version-up (Figure 1.1).

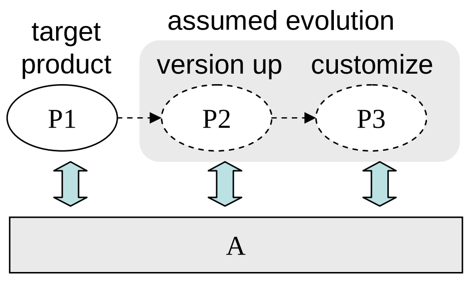


Figure 1.1: Example of Architectural Design for a Single Product

- Designing product-line architecture for a family of products. Here we develop software architecture that will be shared by products in the family (Figure 1.2).

- Examining the scope of product-line architecture, i.e. to determine the set of products that share the same architecture. Here we decide how we divide the set of products into sub groups each of which shares the same architecture (Figure1.3).

1.2 The Problem

As mentioned above, software architecture is quite important and we have to carefully design software architecture so as to make software have required characteristics and make

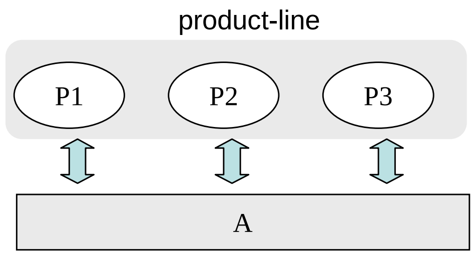


Figure 1.2: Example of Product-line Architectural Design

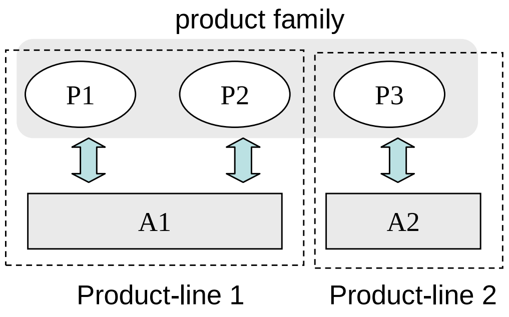


Figure 1.3: Example of Product-line Scoping

software development efficient. However, compared with ordinary software design field, research on software architectural design is not enough and many problems have to be solved.

Though software architectural design has commonalities with ordinary software design, there exist unique characteristics to software architectural design. The followings are typical characteristics for software architecture:

- In software architectural design, we have to examine requirements on functionalities and that on quality attributes. For functionalities, there are some techniques to design software architecture, such as analyzing hot/frozen spots from functionalities, and design software architecture based on the analysis. Similarly, we have to examine requirements on quality attributes so as to design software architecture to be the platform of the software that fulfills the requirement of quality attributes. However, we do not have any systematic architectural design method to handle requirements on quality attributes.

- As software architecture is the fundamental software structure and become the basis of the succeeding software design decisions, we have to design software architecture in the early phase of software development. Therefore we cannot check the appropriateness of the design by developing software actually. Far from that, we do not have enough information to do so. This implies that software architectural design necessarily has the decision-making aspect and the risk-management aspect.

- We need a systematic way to make decisions in architectural design. Architectural design has to be done in the early phase of software development, and we have to make many design decisions based on information we can obtain at that time.

As software architectural decisions are important, we want to make architectural decisions systematically. We also want to make our decision traceable. However, so far, we do not have such a decision-making framework for architectural design.

The objective of the study is to find out a practical method for software architectural design, that reflects the characteristics mentioned above. Here 'practical' means that we can apply the method to real problems. In order to apply the method to real problems, the method has to have the following characteristics:

- We can apply the method to actual size of the problem. Even though experts may be able to design systems by their own ways, systematic methods are still important especially when we design a large and important system, because it is dangerous to design a large system depending upon personal skills. In such a case, it is also required to design the system in an objective way because we have to claim the appropriateness of the design to many stakeholders.
- We can apply the method in a reasonable cost. The method does not use any design technique that requires lots of cost, compared with existing way.
- The method has enough 'resolution' for architectural design. In architectural design, we have to examine multiple alternatives to select most suitable one. The method has to distinguish these alternatives and select suitable one depending on the requirements and design policy.

1.3 Overview of the Solution

In this paper, we define architectural design as selecting most appropriate architecture from multiple options, in terms of requirements and design policy. Figure 1.4 shows an image of software architectural design.

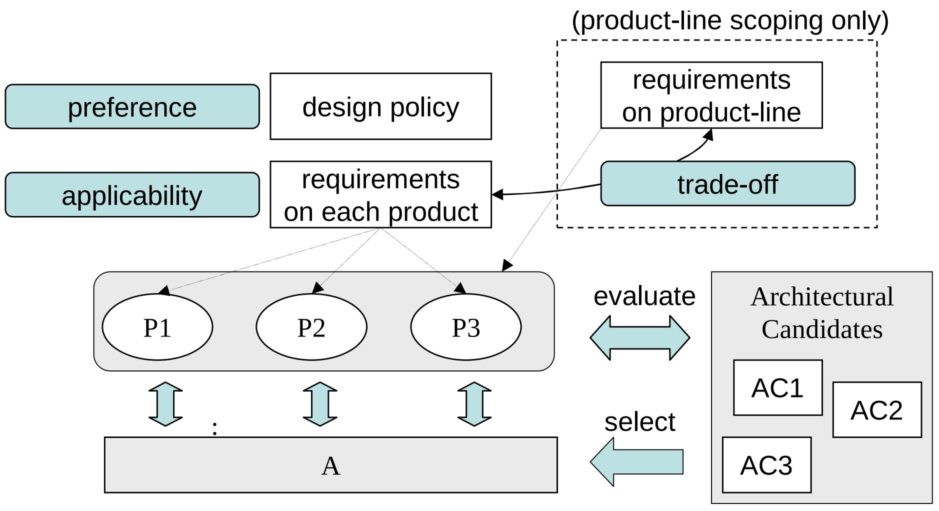


Figure 1.4: Architectural Design

In the figure, we are trying to select architecture from architectural candidates. We have requirements on each product both on functionalities and quality attributes. We also have design policy, in which we define the importance of quality attributes. For example, we may want to make development cost as low as possible, or we may want to make product as reliable as possible. When we examine the product-line scoping, we also have requirements on product-lines, in which we define how we want to develop product-lines. For example, we want to develop the high-end product first, and then release the standard model as first as possible, and so on. In order to select an architecture from candidates, we have to evaluate each architectural candidate, in terms of these requirements and design policy.

In order to examine the architectural design method, we make case study on an actual architectural design project to examine the nature of software architectural design. Based on the observation, we find it important to focus on the followings in architectural design;

- Applicability: If we can attain the requirement on a product based on an architecture, we say that the architecture is applicable in terms of the requirement on the product. In general, when we design architecture, we examine multiple products that will be developed on the architecture. Therefore, we have to select architecture that will be applicable in terms of requirements on corresponding products.
- Preference: There may be multiple applicable architectural candidates. However, these applicable candidates generally have different characteristics; for example, one is good for performance and one is good for cost, and so on. In architectural design, we define design policy in which we determine the importance or weight of each quality attribute, and we select the best one form candidates based on the design policy.
- Tradeoffs: In product-line scoping, we have to consider tradeoffs between benefit for development of each product and benefit for product-lines as a whole. For example, if we are to have a single architecture shared by every product, that is good for development cost, because we only develop a single architecture. On the other hand, If we develop architecture for each product, that is good for each product, however it will be expensive to develop multiple architectures. We have to examine this kind of tradeoffs.

Architectural design is to select architecture considering applicability, preference and tradeoffs. In other words, in architectural design, we have to decide the ordering among architectural candidates in terms of these three concepts.

Figure 1.5 shows the overview of the architectural design technique proposed in this paper. This technique consists of three major parts:

- Examine applicability: Here, we use two techniques; aspect-oriented analysis, and identifying categories of requirements. Aspect-oriented analysis is a technique to separate requirements on each quality attribute from the original requirement. Utilizing this technique, we identify categories of requirements, in which we not only find out applicable candidates, but also identify which one could fulfill widest spectrum of requirements.
- Examine preference: We utilize decision-making techniques to decide the preference in terms of design policy.

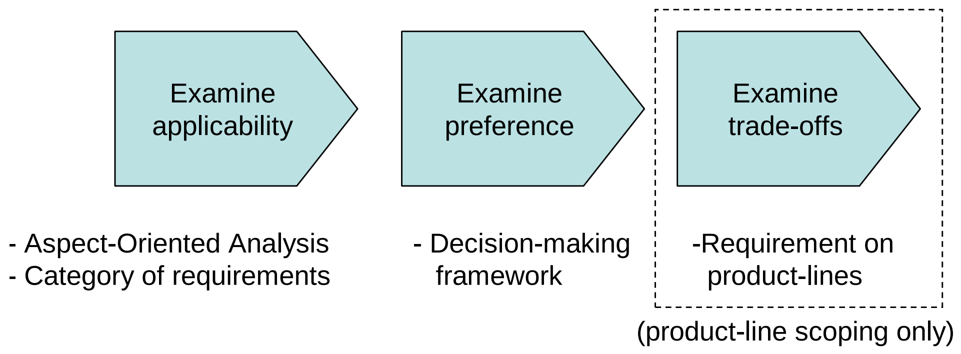

Examine applicability
Examine preference
Examine trade-offs
- Aspect-Oriented Analysis
- Category of requirements
- Decision-making framework
-Requirement on product-lines
(product-line scoping only)

Figure 1.5: Overview of the Architectural Design Method

- Examine tradeoffs: Utilizing the result of previous steps, examine the tradeoff among options. This step is especially important if we design multiple architectures for a product family.

Chapter 2

Related Works

2.1 Software Architecture and Views

Though there are many definitions of software architecture [4, 10, 14, 19, 39, 40, 44] and no single definition is accepted widely, we could say that most of them claim the followings:

- Software architecture refers to the fundamental software structure.
- Software architecture has a strong impact on the characteristics or quality attributes of the software developed on the architecture.
- Software structure governs the entire software design. Once software architecture is determined, it is difficult to change the decision, because many succeeding designs depend on the architecture.
- We can categorize software architecture into relatively independent views. Though there proposed different categorization and views, they have similarity [31, 40, 45].

In this paper, we adopt Ran's definition of software structure and views [40], however, we have adopted the different terminologies.

Some definitions say that software architecture is not only the software structure but also design principle, policy and guidelines. Though these definitions are quite important as these concepts govern the software architectural design, these definitions are little abstract and vague. On the contrast, we just focus on the software structural issues in this paper. Furthermore, in this research, we regard the software architecture as the basis for multiple software products. This makes software architectural design clearer.

We categorize the situation in which we have to design the software structure for multiple products into two. One is designing a single product considering its evolution (modification and version up). In this case, we consider the possible variations of requirements on the products. However, the focus of the design is the target product. The other is designing the architecture for product-lines. In this case, we also design software structure considering multiple requirements. However, in this case, our focus is on the multiple products. If the architecture is good for a product, but it would be not good for other products. We have to examine whether or not the architecture is good for each product in product-line, and maximize the total benefit.

2.2 Architectural Descriptions

There are two different approaches for architectural descriptions. One is to prepare special language for architectural description. Typically, such languages, usually called Architecture Description Language (ADL) [1, 9, 34, 35, 43], have mathematical background to enable mathematical treatment, or have execution semantics to enable simulation. Another approach is to utilize modeling language based on relatively loose formalism, such as Unified Modeling Language (UML) to enable semi-formal description for human communication or simple tool manipulation such as code generation [16, 18, 31]. In this paper, we adopt UML to describe software structure.

ADL is good for analyzing known characteristics that can be modeled on the mathematical foundation. For example, it is good to analyze the performance of the software using the performance model. However, relationship between software architecture and quality attributes is not well understood, and the application of ADL is limited. Especially, in the early phase of software architecture, we have to assess wide spectrum of quality attributes based on limited information. In such a case ADL approach is not appropriate.

Semi-formal languages such as UML [38] can be used in wider situation. As these languages are weak, we can not use them for strict analysis or simulation. However, it could be used to model variety of concepts in a uniform manner. As our focus is making the appropriate architectural decision considering wide and variety of aspects related to software quality attributes, these semi-formal languages are more appropriate.

UML is good for describing the static aspects (information or data) and dynamic aspects (functions and timing). However, we do not have good way to describe requirements on quality attributes using UML. In this paper, we attach these requirements to the modeling components such as class and collaborations. It is not enough in terms of modeling purpose, but it is one of the most common practice adopted in the actual software projects.

2.3 Design Method

As software architecture is the fundamental structure and is the basis of the software design, it is expensive to change the software architecture. That is the main reason why we have to carefully design software architecture.

Most software design methods treat software architectural issues. One of the main issues in treating software architectural issues is development-time quality attributes such as reusability and extensibility. In early 70's, they already argued about the module strength and coupling. We can say that major methods such as structured-method and object-oriented method focus on finding good module structure. However, in these software design method, software structure and software architecture is not clearly distinguished. In real-time structure method proposed by Hatley [17], they treat architectural issues as assignment of functions to components connected by communication path.

Designing the reusable assets such as framework and components requires more explicit focus on software architectural issues. In these design, we do not focus on just a single product, but potential products that will be developed on the reusable assets. In framework design, we are to find hot-spots and frozen-spots in order to design the architecture that can manage the variation of potential products.

In order to develop software and software architecture, it is not enough to design from functional aspects. We have to consider the aspects related to quality attributes. However, as we have mentioned, relationship between software architecture and quality attributes is not well understood. However, there are some researches and practices that are to handle quality attributes. For example, in engineering fields, there are researches that treat quality attributes such as performance [15, 17, 42]. Buhr proposes a method in which they analyze coarse-grained dynamic structure using use-case map [6].

In this paper, we focus on architectural design by examining requirements of multiple products that will be developed on the target architecture. We examine these requirements, especially requirements on quality attributes and find out appropriate design directions. We do not focus on analyzing strict relationship between architecture and quality attributes, but are to clarify the general framework of software architectural design.

2.4 Architectural Design

In actual software development, there are typically two stages where we design software architecture. One is the actual structural design, in which we examine every technical detail and decide the concrete design. In this phase, we have to clarify everything, specification and design, and make architecture clearly designed.

The other is in the early stage of the software development in which we find out the design directions of the software. In this phase, we do not examine every thing, but focus on the significant issues that have to be determined in the early phase. The global architectural style and the basic architectural techniques are examples of issues that have to be decided in this phase. Unless we make any decision on these issues, we cannot design software any further. In these early phase, we may not have enough information for architectural design, however we have to make decision on architecture anyhow. In this paper, we focus on this type of architectural design.

Software design is a creative activity, but we do not always create new design. In software architectural design, we could point out three typical types of architectural design.

- Create new architecture based on the requirements. Structured method based on functional decomposition is one of the way to develop software structure from the requirements.

- Develop architecture based on the pattern. As same as the design pattern [12], there are approaches to define catalogues of architectural styles, to solve problems in architectural design [7, 30, 36, 44]. It is important to understand the characteristics for architectural styles and utilize them in actual architecture, as it is expensive to actually check the characteristics of architectural styles by ourselves.

- Select the architecture that is most appropriate for the requirements from architectural candidates. In the architectural design, it is common to find design directions from multiple options.

In this paper, we focus on the third case, select architecture from multiple options. In selecting architecture, we have to evaluate each architecture in terms of requirements, and decide most appropriate one. Therefore, evaluation techniques are one of the main focus [21, 22]. Though some similar works are categorized into architectural evaluation field, in this paper, we categorize our research into design field, as our objective is to select an appropriate architecture anyhow.

There are another methods that focus on the problem specific to architectural design [30, 8], and we also have to consider these aspects when we design architecture in actual project.

2.5 Product-line Architecture

Product-line architecture is architecture that is shared by members of a product-line. When we are to develop product-lines efficiently, it is important to develop product-line architecture and set up reusable assets based on the architecture. Usually, product-line architecture means, not only architecture itself, but also systems of strategic reuse based on shared architecture [4, 4, 46, 48]. Therefore, non-technical issues such as project management, cost estimation, and risk management are indispensable for product-line architecture. However, in this paper, we focus on technical issues of product-line architectural design.

When we develop product families, we may divide products in the product family into multiple product-lines. Though products in the same product family have some commonalities, it is dangerous to share a single architecture by every member of product family. As product family is determined by many factors such as business issues, technical issues, development- side issues, we have to carefully examine the products and determine the product-lines. This activity is normally referred as software scoping [11, 46], and in usual, it is not considered to be the design activity but business activity to define the product-lines and their development activity. However, in this paper, we just focus on the technical aspect of the software scoping, and categorize the activity in the software architectural design.

Chapter 3

Architectural Design

3.1 Software Architecture

3.1.1 Software Structure

As the basis of our research, we adopt the conceptual framework for software structure proposed by Ran [40]. Though this framework well explains the nature of software structure and the relationship between software structure and quality attributes, it does not explicitly capture the differences between architecture and software structure. In order to extend the framework, we have slightly changed the terminology. Figure 3.1 shows the framework.

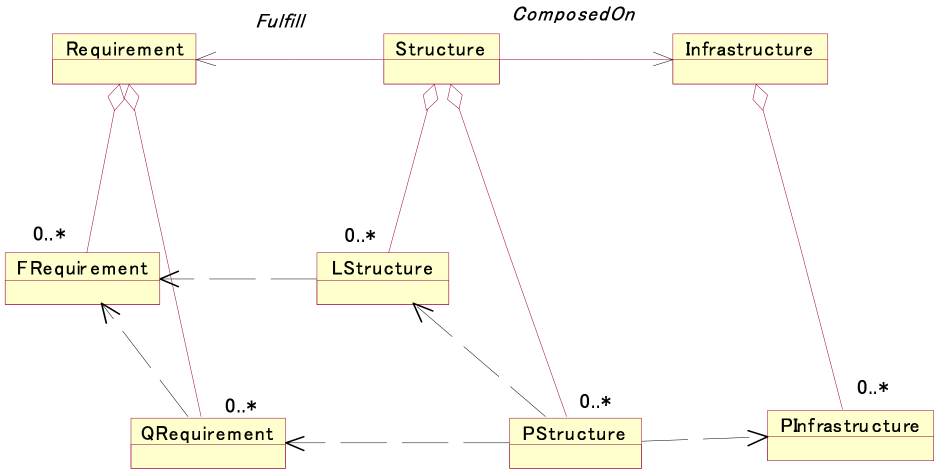


Figure 3.1: Ran's Conceptual Framework — Revised

The following is the summary of the framework[1]:

- *Software structure* (*Structure*) is composed on the infrastructure of software, and is designed so as to fulfill requirements. Requirement is categorized into requirement on function (*FRequirement*) and requirement on quality attributes (*QRequirement*).

[1]In the diagram, we use contracted form for each name due to the space reason. In this case, 'structure' is a concatenate form, and in the text we use 'software structure'.

- *Software structure* is categorized into logical structure (*LStructure*) and physical structure (*PStructure*). *Logical structure* is a special software structure that is constructed on the conceptual components. On the other hand, *physical structure* is constructed on the physical infrastructure (*Pinfrastructure*). There are multiple *physical structures* dependent on the physical infrastructure, such as run-time structure and development-time structure. Each structure is designed to fulfill corresponding requirement on quality attribute (*QRequirements*). For example, run-time structure is designed so as to fulfill requirement on run-time quality attribute such as performance and run-time memory size.

- *Logical structure* represents the important notions in the target domain, and relationship among them. In the ordinary development method, functionalities are represented in this model. *Physical structure* reflects the logical structure and has to attain corresponding requirement on quality attributes.

3.1.2 Software Structure and Software Architecture

We extend the Ran's framework so as to explicitly capture the nature of software architecture. As changes of architecturally significant decisions have a serious impact on succeeding design decisions, we are to design software architecture to be steady throughout software development and evolution. In other word, software architecture has to be the platform of potential software that is assumed to be developed on it. Figure 3.2 shows the conceptual framework for software architecture.

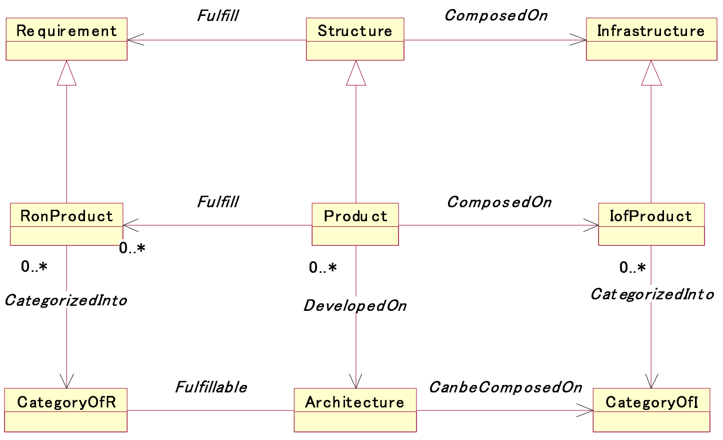


Figure 3.2: Conceptual Framework for Software Architecture

The followings are the overview of the framework.

- Product is constructed on the infrastructure of the product (*IofProduct*), and is designed to fulfill requirement on the product (*RonProduct*). This is the same relationship among requirement, *software structure* (Structure) and infrastructure defined in Ran's framework.

- *Architecture* is the platform of potential products. Here, "potential products" means products that are assumed to be developed on the architecture. In other words, they are products included in the scope of the design.

- As architecture should be the basis for potential products, each product is developed on the architecture and has to fulfill the requirements on the product. If we have a single architecture shared by multiple products, the architecture should fulfill the range of requirements on every product. We call the requirement on architecture that should be the basis of multiple products as category of requirements (*CategoryOfR*). We will explain the idea in 4.3 in detail.
- Infrastructure may change during the evolution or we have to port products onto other infrastructure. We introduce the notion category of infrastructure (*CategoryOfI*), as same as the requirement case.

3.1.3 Assumed Scenario and Product-line

As we explained in 1.1.2, when we design software architecture, we have to examine the requirements on software that will be developed on the architecture. In other words, when we design software architecture, we have to have any information about potential products that will be developed on the architecture. We call these potential requirements as assumed scenario. An assumed scenario is a set of requirements that are considered to be required on software that will be developed on the architecture.

Though it is difficult to identify potential requirements, we try to identify them in actual software development. For example, when we try to develop software spirally, we assume the development scenario in which we decide the items to be developed and the order of development. The development scenario may change, if we encounter problems or if the situation of development is changed. However, we have to assume the scenario to make the strategic development plan. Similarly, when we develop systems, we examine the possible enhancement of the system or change of requirements, in order to handle such expected changes.

When we develop product-lines, we have to assume scenario more clearly. We have to make plan for develop product-line, in which we define what kind of products are included in product-family, what characteristics each product has, and how we develop these products. Without this kind of assumption, we cannot expect the strategic development based on the reuse technology.

3.2 Architectural Design

3.2.1 Definition of Architectural Design

Architectural design is the process in which we find out the structure of components provided by the infrastructures so as to fulfill requirements. In this paper, we define architectural design as the process in which we select most appropriate architecture from architectural candidates in terms of requirements on potential products and design policy.

We explain the intention of the definition.

- In architectural design, it is common to make a list of possible architectural candidates and select most appropriate one from the list. This definition assumes such a situation. This process is one of the typical ways of design, especially when we are to insist the appropriateness of the design decision. On the other hand, it is a little "heavy" process if we have to make a list of candidates for every design decision.

- As we explained in 1.1.2, potential products are products in assumed development/evolution scenario, or planned product-lines.

- Appropriate architecture usually means that selected architecture satisfies requirements. However, in actual development, it sometimes happens that we cannot fulfill requirements by any architectural candidates. That happens because we cannot say that there is no architecture that satisfies requirements until we actually enumerate candidates and assess the characteristics of them. Therefore, we do not exclude such a situation, and in this situation appropriateness is judged by the design policy.

- "Design policy" gives the rules to determine the preference among candidates. Though there may be variety of ways to define "design policy", in this paper, we define design policy by the preferences among quality attributes. For example, assume that we have two architectural candidates; one is good for performance and the other is good for cost. If these candidates satisfy requirements, we cannot determine which one in most appropriate. However, if design policy says that cost in the most important quality attribute, we can select one.

3.2.2 Types of Architectural Design

As we discussed in 1.1.2, we will examine three types of architectural design.

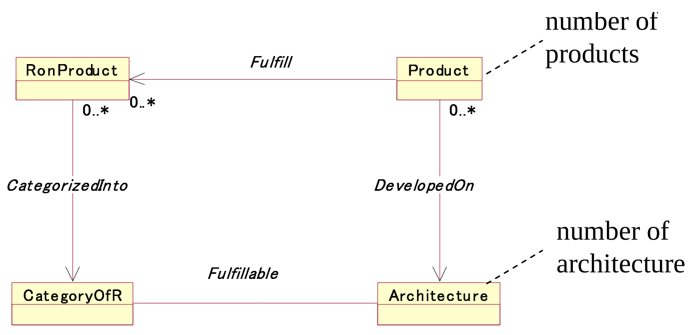


Figure 3.3: Variations of Software Architectural Design

Figure 3.3 shows a part of the conceptual framework shown in Figure 3.2. When we design software architecture, we examine target products that will be developed on the architecture, and "number of products" means the number of the target products. In developing product families, we may design a single architecture for the products in the family, or we may divide products into subsets in which they share the same architecture. "Number of architecture" refers to the number of architecture to be designed. Usually 1 $\leq$ "number of architecture" $\leq$ "number of products" holds. We will explain three types of architectural design shown in Table 3.1.

- Architectural Design: Designing architecture of a product. Here we design software architecture for a product considering the evolution of the software. In this case, requirements include not only requirements on target product but also assumed scenario for the product (Figure 3.4).

Table 3.1: Three Types of Architectural Design

	number of architecture	number of products
Architectural design for a single product	1	1
Product-line architectural design	1	N
Product-line scoping	M	N

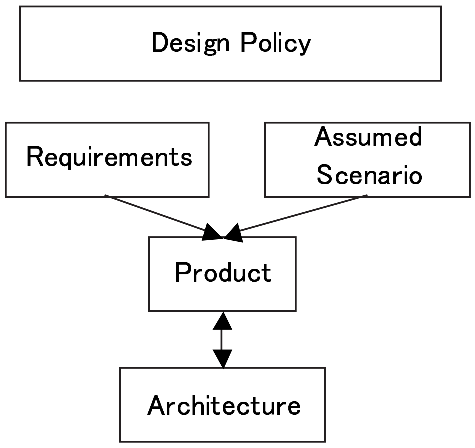


Figure 3.4: Architectural Design for a Single Product

- Product-Line Architectural Design: Designing product-line architecture for a family of products. Here we develop software architecture that will be shared by products in the family. In this case, requirements include requirements on each product in the family, and every product shares the same architecture (Figure 3.5).

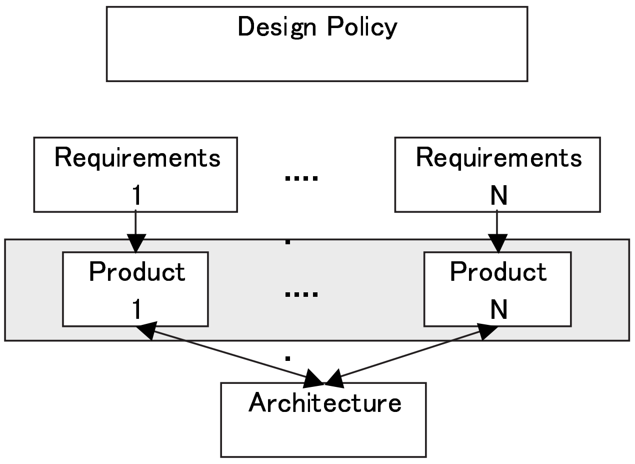


Figure 3.5: Product-line Architectural Design

- Product-Line Scoping: Examining the scope of product-line architecture, i.e. to determine the set of products that share the same architecture. Here we decide how we divide the set of products into sub groups each of which shares the same architecture. In this case, requirements include not only requirements on each product but also requirements on scoping (Figure 3.6).

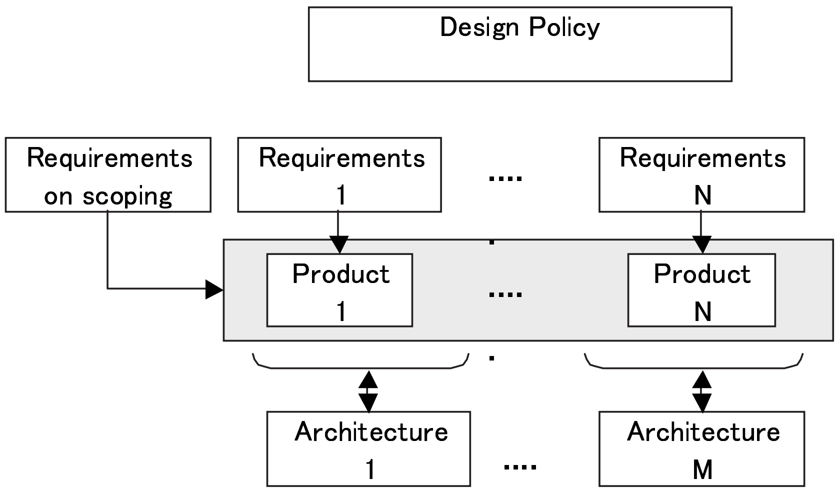


Figure 3.6: Product-line Scoping

Usually, product-line scoping is not called architectural design. However, we assume this is one of the types of architectural design, as this involves quite similar technical issues as architectural design.

3.3 Analysis of the Problem

3.3.1 Overview of the Project

In order to evaluate the technique, we pick up the actual architectural design project, which designed architecture for on-board (in vehicle) systems for ITS (Intelligent Transport Systems) and make observation on the characteristics of architectural design. We also apply our technique to the same problem to demonstrate the applicability of the technique to actual problem.

This project, "Study on ITS on-board system architecture", was completed in 1997 by Association of Electronic Technology for Automobile Traffic and Driving (JSK), in which they investigated the required services and important technologies in the field, and examined the higher-level architecture for on-board system [2].

In the system architecture (SA), they defined 45 sub-services (correspondent to product in our technique), such as route guidance, assistance for economic driving, provisioning of road traffic control, and so on. They analyzed 45 sub-services to identify necessary information and functions using object-oriented analysis. For each function, they enumerated candidates of architecture to realize the function, and qualitatively compared candidates in terms of quality attributes. After that, for each sub-service, they checked every function required for the sub-service, and selected the proper candidate considering the requirements on the sub-service and the characteristics of candidates.

Though ITS are not pure software systems but systems that are realized by software and hardware utilizing communication technology, the differences are not so important in the early stage of architectural design. Firstly, in many systems, we start analysis before dividing the system into software part, hardware part, and human activities. Secondly, when we think of many software systems, most of them are not pure software systems. For example, when we design web application, we have to think of networking issues; when we design an embedded system, we have to consider the hardware configuration. Thirdly, this ITS project have adopted software methodologies, object-oriented method, to the system architectural design, and the basic scheme of the design is almost the same.

The reasons we have picked up this project for the evaluation are the followings; 1) as this project was a large joint project among experts of services, system architects, and specialists of each technologies, the result of the architecture design is considerably reliable, 2) as they have left the reason of architectural selection and basic materials as reports, it is relatively easy to trace their reasoning based on the written documents.

In 3.3.2, we will overview the project and observe the characteristics of architectural design.

3.3.2 Observation from the Project

Though there are hundreds of functions used by 45 sub-services, there are important functions that they call "common functions", which are shared by more than two sub-services. We focus on these common functions, because they are related to commonalities between requirements. Among them, we have picked up 21 important functions, because for these functions, there left enough information in the report. The followings are the observations from analyzing the architectural selection for these 21 functions:

- Though they have considered dozens of quality attributes when they characterize architectural candidates, they use restricted number of quality attributes for each architectural selection.

- Among these quality attributes, some quality attributes are used to judge the applicability (namely, they reject some candidates because the candidates cannot fulfill the requirements on this quality attributes). Other quality attributes are used for relative comparison between candidates to select a "better" candidate among applicable candidates. We call the former as "key quality attribute" in this paper.
- If they cannot select a candidate from key quality attributes, they select a candidate based on the relative comparison. In some case they have examined the characteristics of the sub-service not only from quality attributes but also from knowledge about the sub-services.

Based on the above observation, we have examined how architectural selection has done for these 21 sub-services. Table 3.2 shows the followings: number of candidates for the sub-services, number of quality attributes they used in architectural selection, number of key quality attributes, whether or not factors other than quality attributes are used, and the number of selection patterns, i.e. how many types of reasoning exist. For example, consider that a function F is used by three sub-services S1, S2 and S3, and for S1 and S2 they select candidate (a) as it is good for cost, and for S3 they select candidate (b) as it is high performance, we say the number of selection patterns are two. Even if two sub-services select the same candidate, we say that they are different selection patterns, if the reason is different. In Table 3.2, id such as C1-02 is the id used in the project. This table has 19 functions, as two functions have only one candidate and we ignore them.

Based on the observation and the results summarized in Table 3.2, we consider the followings about the applicability of our techniques:

- In this project, 11 sub-services out of 19 have key quality attributes and we can identify two or more categories for them. 4 out of 11 have two key quality attributes. This means that in architectural selection it is important to judge the applicability of candidates in terms of requirements.
- If there are no key quality attributes, all candidate fulfill the requirements. In this case, they select an architecture using some technique to decide the relative preference. (In this project, they use scoring method).
- Many of the architectural selection (13 out of 19) can be done just using quality attributes. This means that in architectural selection, in many cases, requirements on quality attributes have enough information for architectural selection.
- Many of the architectural selection (12 out of 19) have more than two selection patterns. It becomes difficult to clarify the reason of architectural selection, if the number of the patterns becomes large. In our technique, we can hierarchically (step-wisely) select the architecture that makes the reasoning clearer.

As shown above, we can observe that in architectural selection, it is important to determine "applicability" and "relative preference" for architectural candidates.

Table 3.2: Overviews of Architectural Selection for ITS Project

	number of candidates	number of quality attributes	number of key quality attributes	other factors are used or not	number of selection patterns
C1-02	9	4	1	no	5
C1-03	6	4	2	yes	6
C1-04	4	3	0	yes	1
C1-05	3	2	0	no	1
C1-06	2	1	0	no	2
C1-07	2	3	1	no	3
C1-08	4	4	2	no	1
C1-09	4	3	1	yes	2
C1-10	3	2	0	no	2
C1-11	3	5	0	no	3
C1-13	3	4	0	no	2
C1-14	3	4	0	no	1
C1-16	4	3	1	yes	1
C1-17	5	3	1	no	1
C2-02	2	1	1	no	2
C3-01	4	6	2	yes	3
C3-02	2	3	0	no	2
C3-04	2	3	1	no	1
C4-01	2	2	2	yes	3

3.3.3 Observation on Product-line Scoping

In examining product-line scoping, we have to consider different aspects from that of single product architectural design and product-line architectural design.

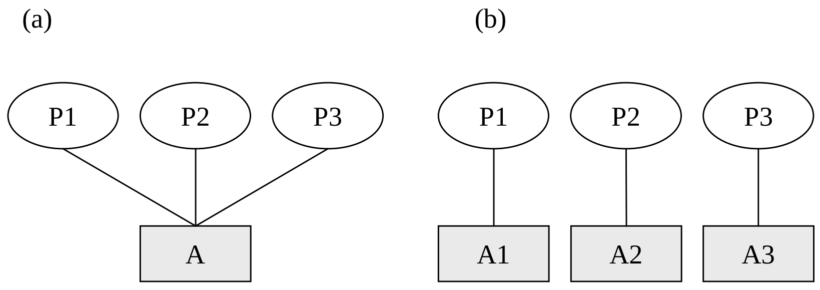


Figure 3.7: Two Extreme Situations in Product-line Scoping

Figure 3.7 shows two extreme situations in product-line scoping.

- In (a), every product shares the same architecture, and we can expect to reduce the total development cost as we develop a single architecture for every product. However, it may not be good for the characteristics of each product, as it is difficult to provide a single architecture that is best for every products.
- In (b), each product has its own architecture. In this case we could develop specialized architecture for each product to realize the best characteristics. However, it is expensive and bad for the total development cost.

As observed above, in product-line scoping, we have to examine not only the characteristics of each product but also the characteristics of total development, such as development cost and reuse ratio. These two characteristics are tend to conflict each other, and in such a case, we have to examine the tradeoffs between these two.

3.4 Modeling Framework

3.4.1 Model for Software Structure

In this section, we explain the modeling framework for software structure used in the paper. Though the design method proposed in the paper does not tightly combined with the modeling framework below, it is incorrect to insist that the method is free from modeling method because there are variety of modeling techniques for software structure. Our modeling framework is one of common frameworks based on UML. Figure 3.8 shows the framework.

- Logical structure (LStructure) and physical structure (PStructure) in the conceptual framework (Figure 3.1) are modeled as logical model and physical model respectively. These two models are corresponding to analysis model and design model in ordinary methodology.

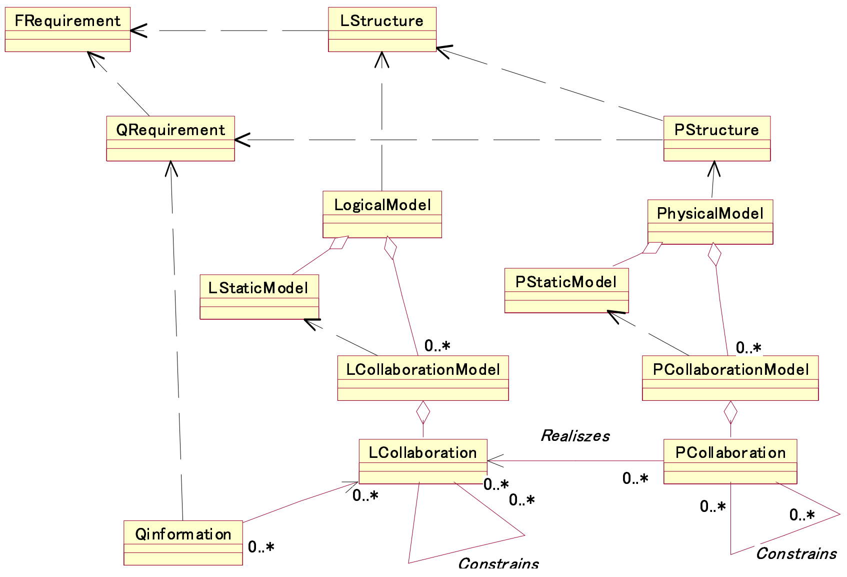


Figure 3.8: Modeling Framework for Software Structure

- Logical model consists of logical static model (LStaticModel) and logical collaboration model (LCollaborationModel). Logical static model is described by means of class diagram of UML, and logical collaboration model is described by means of collaboration diagram of UML. Physical model also consists of physical static model (PStaticModel) and physical collaboration model (PCollaborationModel).

- Logical model is obtained by analyzing the requirements on function. In logical static model, there defined conceptual elements that are used in logical collaboration model. In logical collaboration model, there described interactions among components and roles played by them, typically for specific aspects of software behavior. In UML, there are two types of collaboration model, one is at specification level and the other is at instance level, and we believe both of them are useful for our purpose.

- As we have explained in 3.1.1, there are multiple physical structures dependent on the different infrastructures. Corresponding to these multiple physical infrastructures, we can define multiple physical models that describe different structures, such as run-time structure and development-time structure.

- We do not model requirements on quality attributes straightforwardly. We describe requirements on quality attributes as added information to logical model; attached information to components in logical static model or logical collaboration in logical collaboration model. We will explain this further in 3.4.2

- Logical collaboration model and physical collaboration model consist of logical collaborations and physical collaborations respectively. If physical structure satisfies the requirements on products, there must be physical collaborations corresponding to logical collaboration that satisfies both requirements on function and requirements on quality attributes.

3.4.2 How to Represent Requirements on Quality Attributes

Though our modeling framework is almost as same as modeling framework adopted by typical object-oriented method, the way to express requirements on quality attributes is slightly extended. In this section, we explain how to represent variety of requirements on quality attributes in this modeling framework.

- As we are interested in the quality attributes that relate to software architecture, we categorize quality attributes by means of types of software architecture that determines the quality attributes. Namely, we categorize quality attributes into run-time quality attributes, development-time quality attributes, deployment-time quality attributes, and so on. Based on the category, we attach each requirement on quality attributes to corresponding software structure model. For example, requirements on run-time quality attributes are attached to run-time structure model, requirements on development-time quality attributes are attached to development-time structure, and so on.

- When we attach requirements to software structure, as explained above, we attach the requirements to most appropriate modeling components. Typically, requirements are attached on static components modeled in logical static model or dynamic collaborations modeled in logical collaboration model. As both components and collaborations can be defined hierarchically, we can attach requirements at arbitrary level.

The way how to attach the requirements on quality attributes to proper components or collaborations is context dependent, i.e. we have to analyze the required quality attributes and judge most appropriate target. Table 3.3 shows some examples of quality attributes picked up from ISO 9126, and typical examples how to represent the requirements. We do not insist that all the quality attributes described here have strong relationship with software structure, but we show the table to clarify how to represent requirements on quality attributes. Basically, if the quality attribute is about static characteristics of components, it is attached to the components. If it is about functions, it is attached to the collaborations representing the functions.

In order to clarify our intention, we explain some quality attributes described in the table.

- Functionality: Suitability is about the quality of design, and can be attached to collaborations at design level. Interoperability typically relates data format and is about run-time component structure.

- Reliability: Typically this quality is about set of services (functions), and can be attached to collaborations that include the functions.

- Usability: Typically this attribute is about human interface or human operation and this relates to run-times static / dynamic structure because that affects user operation.

- Efficiency: Time behavior relates to run-time collaboration. Resource behavior relates to run-time collaboration (run time resource) or run-time / development-time components structure (static resource).

Table 3.3: Quality Attributes and their Representation

Quality characteristics	Quality sub-characteristics	Software structure	Components/ collaboration
Functionality	Suitability	Run-time	Collaboration
	Accuracy	Run-time	Collaboration
	Compliance	Any structure	Depends on the standard
	Interoperability	Run-time	Collaboration
	Security	Run-time Deployment-time	Both
Reliability	Maturity	Run-time	Collaboration
	Fault tolerance	Run-time	Collaboration
	Recoverbility	Run-time	Collaboration
Usability	Understandability	Run-time	Both
	Learnability	Run-time	Both
	Operability	Run-time	Both
Efficiency	Time behavior	Run-time	Collaboration
	Resource behavior	Run-time Deployment-time	Both
Maintainability	Analyzability	Development-time Run-time	Both
	Changeability	Development-time	Component
	Stability	Development-time	Component
	Testability	Development-time Run-time Deployment-time	Both
Portability	Adaptability	Development-time	Component
	Installability	Development-time	Component
	Conformance	Any structure	Depends on the standard
	Replaceability	Development-time	

- Maintainability: Analyzability relates to how easy to understand the dynamic behavior of software. Changeability and stability relate to development time static structure. Testability depends on both structure.

- Portability: Basically relates to development-time component structure.

Chapter 4

Design Techniques

4.1 Overviews

4.1.1 Applicability and Preference

As observed in the previous chapter, we utilize applicability and preference in selecting an architecture from architectural candidates. Figure 4.1 shows conceptual explanation of these two concepts.

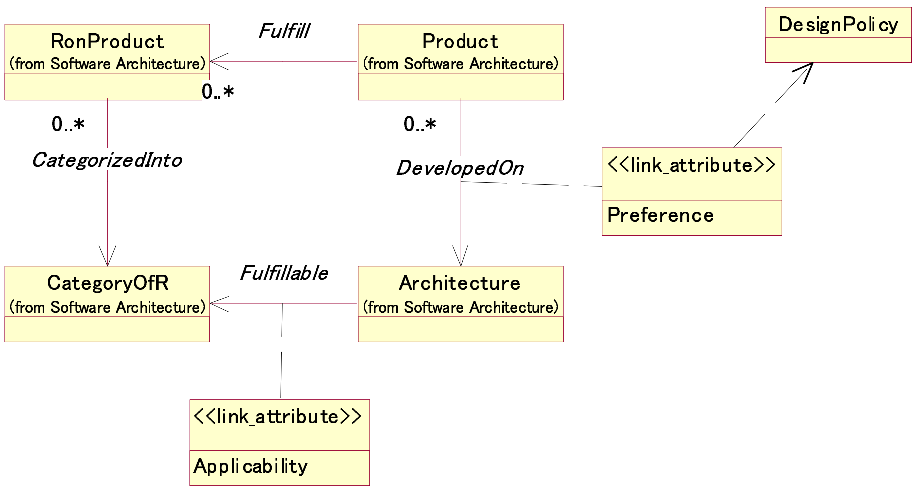


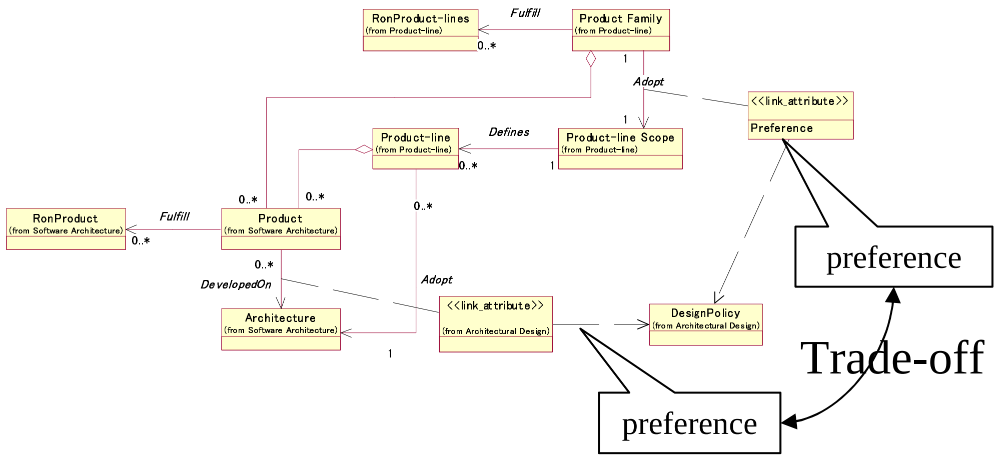
Figure 4.1: Conceptual Model for Applicability and Preference

- In architectural design, we are to select applicable architecture, namely architecture that fulfills the requirements on the products that will potentially be developed on the architecture. In architectural candidates, some architectures are applicable and some are not applicable. We can define applicability for every architecture in terms of requirements on architecture (category of requirements), and this shows to what extent the architecture fulfills the requirements in the category.
- Among multiple applicable architectures, we are to select an architecture that is most preferable in terms of the design policy. We can define preference for every

architecture in terms of product, and this shows to what extent the characteristics of the architecture are consistent with the design policy.

4.1.2 Tradeoffs between Individual Optimal and Whole Optimal

Figure 4.2 is the extended conceptual model, in which we express product-line, product-line scope, product-family, and requirements on product-line.



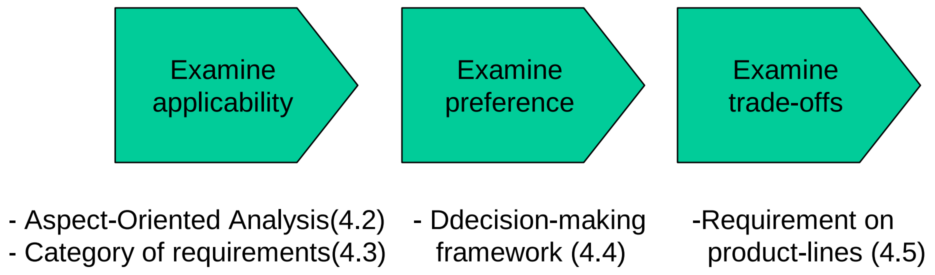
RonProduct-lines (from Product-line)
Fulfill
Product Family (from Product-line)
0..*
1
Adopt
<<link_attribute>>
Preference
1
Product-line (from Product-line)
Defines
Product-line Scope (from Product-line)
0..*
1
0..*
0..*
0..*
RonProduct (from Software Architecture)
Fulfill
Product (from Software Architecture)
0..*
preference
0..*
DevelopedOn
Adopt
Architecture (from Software Architecture)
<<link_attribute>>
(from Architectural Design)
DesignPolicy (from Architectural Design)
1
Trade-off
preference

Figure 4.2: Conceptual Model for Tradeoffs

In product-line scoping, we define multiple product-lines in product family. Here, product-line is a set of products that share the same architecture. As we have observed previously, product-line scope affects the characteristics of the development of the product-family as a whole. Requirements on product-line is the requirements on the product family development. We will explain this further in 5.4.

4.1.3 Overview of Architectural Design

In architectural design, we select an architecture considering the applicability, appropriateness, and tradeoffs. In order to support architectural design, we propose techniques for examining them.



Examine applicability
Examine preference
Examine trade-offs
- Aspect-Oriented Analysis(4.2)
- Category of requirements(4.3)
- Ddecision-making framework (4.4)
-Requirement on product-lines (4.5)

Figure 4.3: Overview of the Architectural Design Method (Revised)

- In 4.2, we introduce Aspect-Oriented Analysis (AOA), that is the technique to separate the requirement on each quality attribute from original requirements. As it is difficult to handle quality attributes at the same time, this technique enables us to analyze requirements one by one. This technique is used in identifying category of requirements, introduced in 4.3
- In 4.3, we introduce a technique to identify category of requirements, using AOA. This technique is used to decide the applicability of the architecture in terms of requirements. This technique can be used to decide the most applicable architecture among options; in other words, it decides the order among options in terms of range of applicability.
- In 4.4, we introduce how we can apply decision-making technique to architectural design. We use the technique to decide the preference among candidates. In our framework, we can decide the preference consistent with the design policy.
- In 4.5, we introduce the notion of characteristics of product-line. In architectural design, we have to consider the tradeoffs among individual optimal (whether architecture is good for each product) and whole optimal (whether architecture is good for product-line as a whole). In order to analyze the tradeoff, we analyze the requirement on product-line and examine the characteristics of product-line as a whole.

4.2 Aspect-Oriented Analysis [25, 27]

4.2.1 Designing Architecture Considering Quality Attributes

As we have discussed in 2.1, software structure has to fulfill not only requirements on functions, but also requirements on quality attributes.

Most design methodologies deal functional design. In structured method, functions are modeled as transformation of data, and modeled in terms of data flow diagram in analysis phase. In design phase, components are structured so as to reflect the data transformation. In object-oriented method, functions are modeled as collaborations and we configure components so as to realize the collaborations. Our modeling framework proposed in 3.4 is categorized into the latter type.

In architectural design, we have to capture "categories of requirements" and "categories of infrastructures", and design software structure as the basis for potential software developed on it. For functional design, we have many techniques to design platform such as inheritances mechanism, framework technologies, and hot / frozen spot analysis, and so on.

In this paper, we do not explain how to analyze functions and design architecture so as to realize required functions. We take a position that we use existing method for functional design. If we do not have to consider requirements on quality attributes, functional design works well. However, if we have to consider requirements on quality attributes, functional design may not work well, as most methodologies consider restricted kinds of quality attributes, typically extensibility and reusability, and do not consider other quality attributes.

It is obvious that even if architecture fulfills requirements on functions, it cannot be a platform if it does not fulfills requirements on quality attributes. Software structures for mission critical systems and ordinary business systems have different structures, even if required functions are exactly the same. In this sense, it is quite important to design architecture considering functions and quality attributes. As there are many researches on functional design, in this paper, we focus on the latter.

4.2.2 Problems

As we mentioned in 4.2.1, though there are multiple methods that handle functional requirements, we do not have enough design methods that handle quality attributes. The basic reason is that we do not correctly understand how software structure affects quality attributes. We just understand them from our experiences.

The followings are typical problems in software design considering quality attributes:

- Generally, relationships among software structure and quality attributes are not well understood. Though there are some researches that investigate on relationship between specific quality attribute and software structure, we do not have any systematic way to design software structure from requirements on quality attributes.
- We have to consider multiple quality attributes, some of them relate to different structure and some relate to the same architecture. It makes requirements on quality attributes large and complicated. We need a systematic way to analyze these requirements and make architectural design easier.

4.2.3 Overview of the Approach

In order to solve the problem, we have adopted the following approaches:

- For each quality attribute, we experientially know that there are important factors that determine the quality attribute. For example, when we examine performance of data retrieval, we focus on important factors such as size of search space, characteristics of data and search type, and so on. We believe that utilizing these factors for architectural design is good for establishing a systematic way for architectural design considering quality attributes.
- As it is difficult to examine multiple quality attributes at the same time, we try to examine the requirement on each quality attribute independently. If we could separate requirements on each quality attribute from original requirements, analysis work is expected to be easier, because the amount of requirements we have to analyze at a time becomes small. We can also expect that we could examine the characteristics of requirements on each aspect easily. This technique is so called ”separation of concerns” that is used when we want to reduce the size and complexity of the original problems [23, 37].

4.2.4 Factors and Aspects

In order to examine systematic way of architectural design considering quality attributes, we utilize factors that are experientially known to determine quality attributes. Here, factors are defined as important determinants of some software characteristics. We observe that we utilize factors when we design architecture. For example, when we design architecture for data retrieval considering performance, we may use factors such as "size of search space" and "types of data retrieval (sequential and random)": if data space is small, exhaustive search will be enough, if data space is large and we randomly retrieve data, hashing mechanism works well, and so on.

Consider two quality attributes, performance of data retrieval and run-time memory size of data retrieval; the performance is determined by data size, data number and retrieval type, and the run-time memory size is determined by memory management policy (what part of the data should be on memory) and data size. In this case these two quality attributes have different sets of factors but some of them (in this case data size) are shared. In general, if we are examining multiple quality attributes, we could find factors for each quality attribute, and some of them are shared. Figure 4.4 depicts the relationship among these concepts.

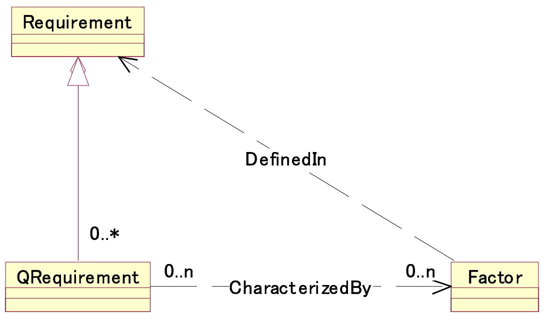


Figure 4.4: Relationship between Requirements and Factors

Here, requirement consists of multiple requirements on quality attributes, and each requirement on quality attribute is characterized by factors. These factors are defined in terms of requirements. For example, if we are interested in size of search space, we have to express this in terms of the notion defined in requirements.

Aspect is a view from which some software characteristics are well captured. If we are interested in the aspect from which characteristics related to some quality attributes are well captured, the aspect can be considered as a view in which factors related to corresponding quality attributes are well described.

Our intention is to separate aspect that relates to specific quality attribute from original requirements. This is regarded as the operation in which we focus on some quality attributes, identify factors related to the quality attributes and define requirements in terms of related factors.

4.2.5 Analysis Method

Here we introduce Aspect-Oriented Analysis (AOA). AOA is a method by which we separate requirements on each quality attribute from original requirements. The followings are analysis steps of AOA:

1. Determine quality attributes: decide the quality attributes we are to examine.

2. Determine components or collaborations related to requirements on quality attributes: As we have mentioned in 3.4.2, requirements on quality attribute are attached onto components or collaborations. We decide components or collaborations to be analyzed.

3. Enumerate important factors: For each quality attribute, make a list of important factors in this context.

4. Characterize components / collaborations in terms of factors.

5. Separate each aspect: For each quality attribute, separate requirements related to the quality attribute from original requirements, utilizing factors related to the quality attributes.

This operation is similar to projection of relational database. When we separate an aspect from original requirements, we focus on factors that relate to the aspect, ignoring other factors. As a result, we can focus on factors we are interested in. When we separate aspect, some requirements become the same, because we may ignore factors that make these requirements different. Consequently, number of requirements may become small (Figure 4.5).

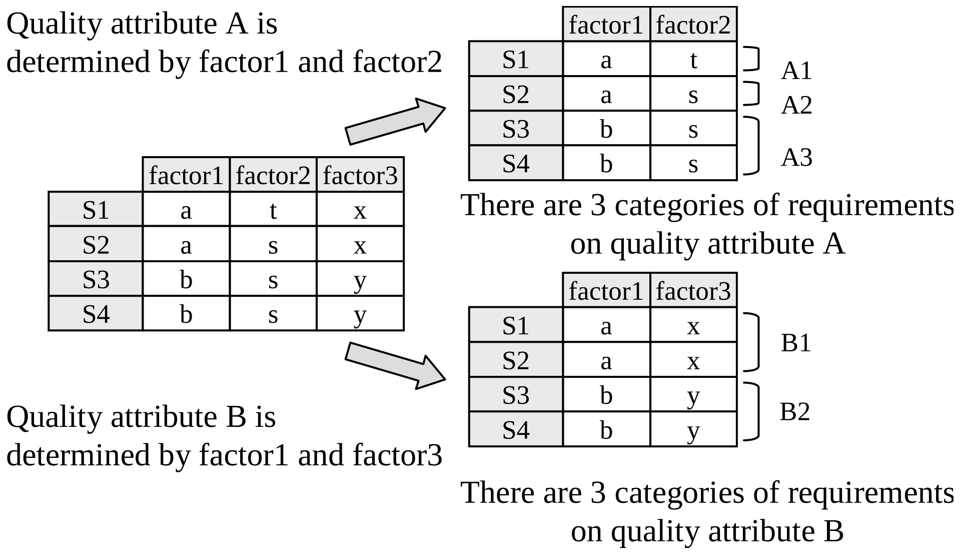

Quality attribute A is determined by factor1 and factor2
factor1
factor2
factor3
S1
a
t
x
S2
a
s
x
S3
b
s
y
S4
b
s
y
A1
A2
A3
There are 3 categories of requirements on quality attribute A
Quality attribute B is determined by factor1 and factor3
B1
B2
There are 3 categories of requirements on quality attribute B

Figure 4.5: Separate Aspects

4.2.6 Example

We explain the method using an example. The system we are to analyze is software for information terminal that retrieves map information stored in CD-ROM database. We have to be careful to attain required performance of data retrieval because CD-ROM drive is low-speed. We also have to be careful on the memory size, because this is embedded software. Each market such as Europe, U.S. and Japan has different standards for CD-ROM format.

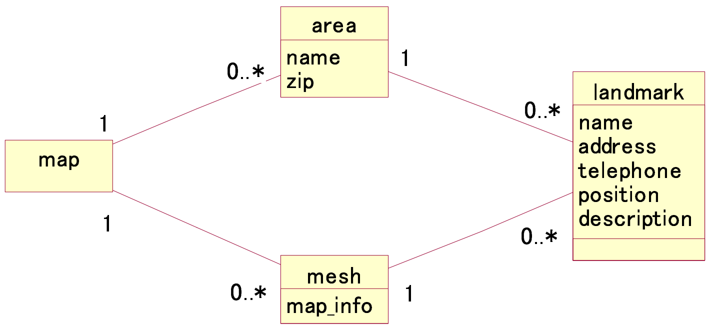


Figure 4.6: AOA Example: Logical Structure of Map Information

Figure 4.6 shows the logical structure of map information. Map has multiple areas, and each area has multiple landmarks. The map is also divided into multiple meshes (square areas), and landmark also belongs to a mesh. Each mesh has mapinfo, large and complex data, which is necessary for displaying map on the screen and deciding the position of the point where user selects by cursor. Description is textual information of landmark, which is unfixed size and some of them are quite large size. Other attributes are small and fixed size data.

1. Determine quality attributes: We design the software considering two quality attributes, response time for data retrieval and run-time memory size.

2. Determine collaborations related to requirements on quality attributes: As response time and run-time memory size in this case relate to functions, requirements on these quality attributes are attached on collaborations. Table 4.1 shows list of collaborations (S1 - S5) and attached requirements on these quality attributes. As the function is data retrieval, we explain the input and output of the data retrieval in the table.

Table 4.1: AOA Example: Requirements on Quality Attributes

	input	output	response	size
S1	zip	name	< 1 sec	< 250K
S2	tel	name	< 1 sec	< 250K
S3	name	tel	< 1 sec	< 250K
S4	position	name	< 1 sec	< 250K
S5	string	name	< 1 sec	< 250K

3. Enumerate important factors: For each quality attribute, make a list of important factors that determine the quality attribute. (Table 4.2)

4. Characterize collaborations in terms of factors: Characterize each collaboration by factors listed in the previous step. (Table 4.3)

5. Separate each aspect: Separate aspect from the requirements. If we are interested in response time, we focus on the factors such as pattern, number, and size. If we

Table 4.2: AOA Example: Factors for Each Quality Attributes

	factors	meaning
response	access pattern	*random* or *sequential*
	number	number of data in search space
	size	*large* or *small* / *fixed* or *unfixed*
size	mapinfo	size of mapinfo in CD-ROM

Table 4.3: AOA Example: Results of Characterization

	pattern	number	size	mapsize
S1	random	100	small/fixed	0
S2	random	100,000	small/fixed	0
S3	random	100,000	small/fixed	0
S4	random	100,000	small/fixed	50K*4
S5	sequential	100,000	large/unfixed	0

are interested in memory size, we focus on factors such as mapsize. Ignoring other factors, we obtain the following separated requirements. (Table 4.4, Table 4.5)

Table 4.4: AOA Example: Separated Aspects (Performance)

	pattern	number	size	response
CP3	random	100	small/fixed	< 1 sec
CP4	random	100,000	small/fixed	< 1 sec
CP5	sequential	100,000	large/unfixed	< 5-6 sec

4.2.7 Application of the Method

Using the technique, we can separate aspects, each of which well captures the requirements on some quality attributes. Namely, we can separate three types of requirements on response time, two types of requirements related to memory size, from original requirements. We can expect that analysis work for requirements on each quality attribute becomes easy, because complexity and size of the requirements we have to analyze at a time becomes simple and small. We can understand the spectrum of requirements on quality attributes in a systematic and analytic way.

This method is a basic technique and it does not help architectural design straightforwardly. In the next section, we introduce another technique for architectural design, in which we fully utilize AOA.

Table 4.5: AOA Example: Separated Aspects (Size)

	mapsize	size
CS1	0	< 250K
CS2	50K * 4	< 250K

4.3 Identifying Category of Requirements [28]

4.3.1 Category of Requirements

In order to design software architecture to be steady throughout evolution, it is important to identify "category of requirements". Here, category of requirements means the set of requirements that can be fulfilled by the same architectural technique. For example, consider the series of products that have the similar functions, such as data retrieval, but the requirements on number of data in the search space becomes larger; say 1,000 items, 10,000 items, and 100,000 items. Assume that we can realize these products on the same architecture until the requirements become 10,000 items, but we need entirely different architecture for 100,000 items. In such a situation, we can say that the first two requirements (1,000 items, and 10,000 items) are in the same category, and the last requirement (100,000 items) is in a different category. The reason why we introduce the idea of "category of requirements" is that, in architectural design, it is important to identify the commonalities and differences of necessary architectural techniques, because if two requirements can be fulfilled by the same architectural technique, we do not need to change the architecture even if they are not exactly the same.

4.3.2 Identifying Category of Requirements

As we mentioned in the previous section, "category of requirements" depends on architectural technique that can be used for the target software. Therefore, when we identify the categories, we have to examine both requirements and possible architectural techniques. The followings are the steps to identify "category of requirements". As category of requirements is defined for each quality attribute, we consider just one quality attribute in this section. In the next section, we will show how to handle multiple quality attributes based on the categories for each quality attribute.

Here product means software we are to consider. Each product includes important functions for which we have to examine architectural techniques in order to fulfill requirements. In general, each product has multiple important functions. In such a case, apply the following steps to each important function.

1. Decide the quality attribute to be examined: Pick up quality attributes we are to examine. For example, in the case of developing products that retrieve data in storage, the response-time of data retrieval and run-time memory size are typical quality attributes to be examined.

2. Identify the variation of requirements in terms of the quality attribute: Examine the requirements on the quality attribute, and identify the variations. For data retrieval example, consider that there are three products in a family, and requirements on

response time are the same, but the number of items in search space is different (Table 4.6). In this case, we can identify three variations in requirements; R1, R2 and R3.

Table 4.6: COR Example: Variation of Requirements on Quality Attributes

	response time	number	
P1	< 2-3 sec	100	—R1
P2	< 2-3 sec	1,000	—R2
P3	< 2-3 sec	100,000	—R3

3. Make a list of candidates of architectural techniques: Considering the variation of requirements, make a list of candidates of architectural technique. For data retrieval example, if the requirements are not severe, we can attain them without considering any architectural technique, and we may design architecture so as to reflect the structure of analysis model that captures the nature of target domain, because it is considered to be good for extensibility. However, if the requirements are severe, we have to examine architectural techniques. In this case we enumerate two candidates; A1 has caching mechanism, and A2 loads index data of storage onto memory at the initialization time. Let A0 be the architecture that reflects the structure of analysis model (Table 4.7).

Table 4.7: COR Example: Candidates of Architectural Techniques

A0	Architecture that reflects the structure of analysis model.
A1	Architecture that adopts caching mechanism.
A2	Load index data onto memory at the initialization time.

4. Identify the categories: Check if each candidate listed in the previous step can be applicable to each variation of requirements. If the set of applicable architectural techniques are the same, categorize the requirements into the same category. If the set of applicable architectural techniques are different, categorize them into the different categories. In the data retrieval example, we determine that we can fulfill R1 and R2 without any architectural technique, but we need A1 or A2 to fulfill R3. Based on this examination, we identify two categories: CP1 that can be fulfilled by A0, A1 or A2 and CP2 that can be fulfilled by A2 and A3. (Table 4.8).

Table 4.8: COR Example: Identifed Categories of Requirements

	requirements	applicable architectural techniques
CP1	R1, R2	A0, A1, A2
CP2	R3	A1, A2

4.3.3 Analyzing Commonalities and Differences

In architectural design, we are to design software architecture to be steady throughout the evolution, or we are to make the architecture to be the platform of the software in the product family. In order to design software architecture to have such characteristics, we have to analyze the commonality and differences of necessary architectural techniques and design software architecture so as to accommodate the commonality and differences.

In the previous section, we propose the technique to identify the "category of requirements", in which we focus on one quality attributes. In actual architectural design, it is common that we have to consider multiple quality attributes; however, considering multiple quality attributes at a time is not easy. We have adopted "separation of concerns" to this problem. Namely, we firstly examine the "category of requirements" that is defined for each quality attributes, and then merge the result to find out the commonalities and differences of necessary architectural techniques.

1. Define the scope of the analysis: Assume the scenario of evolution or that of development of products in a family. Pick up target products to be analyzed from the scenario. We also decide quality attributes, from which we analyze requirements. For example, consider products that retrieve data in storage. We analyze three products P1, P2 and P3 in a product family. We focus on the response time and memory size. Table 4.9 shows the requirements on the quality attributes for these products.

Table 4.9: COR Example: Requirements on Products

	number	response time	memory size
P1	100	< 2-3 sec	small
P2	1,000	< 2-3 sec	
P3	100,000	< 2-3 sec	

2. Identify the "category of requirements" for each quality attribute: Using the technique proposed in the previous section, identify categories of requirements for each quality attributes. Table 4.10 and Table 4.11 show the categories of requirements we have identified for the data retrieval example. In this table, '*' represents "no requirements". The categories of requirements from performance are as same as we have identified in the previous section. For memory size, we cannot use A2 if memory size is required to be small, because loading index data requires considerably large memory size.

Table 4.10: COR Example: Category of Requirements from Performance

	number	response	architectural techniques
CP1	100-1,000	< 2-3 sec	A0, A1, A2
CP2	100,000	< 2-3 sec	A1, A2

3. Analyze commonalities and differences: Firstly, we determine the applicability of architectural technique to each product. Table 4.12 shows how to determine the

Table 4.11: COR Example: Category of Requirements from Size

	memory size	architectural techniques
CS1	small	A0, A1
CS2	*	A0, A1, A2

applicability. From performance aspect, we can apply A0 to P1 and P2, and from size aspect, we can use A0 to every product. Therefore, if we are to attain both requirements, A0 is applicable to P1 and P2. Similarly, A1 is applicable to every product, and A2 is applicable to P2 and P3. Table 4.13 is a matrix obtained from the merged result. In this table, 'x' indicates "applicable.

Table 4.12: COR Example: Commonalities and Differences

	A0	A1	A2
performance	P1, P2	P1, P2, P3	P1, P2, P3
size	P1, P2, P3	P1, P2, P3	P2, P3
merged result	P1, P2	P1, P2, P3	P2, P3

Table 4.13: COR Example: Applicability Matrix

	P1	P2	P3
A0	x	x	
A1	x	x	x
A2		x	x

Based on the analysis, we can understand the commonalities and differences of necessary architectural techniques. Table 4.13 shows what architectural techniques can be applicable to each product. P1 and P2 can use A0, but P3 cannot. Every product can use A1. P2 and P3 can use A2, but P1 cannot. Note that, applicable means that the requirements can be fulfilled, but it does not mean that every applicable architecture can attain the same level of quality attribute. For example, both A0 and A1 is applicable to P1, we can expect that A1 attains better performance than A0, and so on.

4.3.4 Applying the Techniques to Architectural Design

We can apply the technique introduced in the previous sections to early phase of architectural design in which we make decision on the direction of architectural design. In the next chapter, we will explain how to design architecture in detail. Here, we explain typical steps:

1. Determine the style of global architecture: Our technique proposed above is used to select architectural candidates for important functions included in products. In general, each product includes multiple important functions for which we have to examine the architectural techniques. However, architecture of the entire product is

not just a gathering of architectural techniques selected for each important function. In order to make the architecture consistent, we generally examine the architectural style that governs the entire product. This style constrains the architectural selection in the following steps.

2. Identify categories of requirements: Decide quality attributes we have to examine, then, for each quality attribute, identify categories of requirements (see 4.3.2).

3. Analyze commonalities and differences: Based on the identified categories of requirements, analyze the commonalities and differences on necessary architectural techniques (see 4.3.3).

4. Determines the basic policy: Determine the basic policy of architecture selection. In step 3, we have identified applicable architectural techniques to each product. In general, we can expect that the number of architectural techniques decreases by step 3, but we cannot always select one architectural technique. (If the number of applicable architectural technique becomes zero, it means we cannot think of how to fulfill requirements). At this step, we do not consider which architectural technique is good for each product, but examine the product-wide strategy of development. Typically we have to examine from the following viewpoints:

   - If products are developed sequentially, we have to consider if architectural change is necessary or not. In the example of data retrieval system (Table 4.13), if we choose A2, no architectural change is required. However we may intentionally abandon continuation at some point, and develop high-performance model based on different architecture.
   - If products are in the product family, we have to consider to what extent we are to share the same architecture. If we are to reduce development cost, it may be good to maximize the sharing. However, we may want to use different architecture for high-end product, to realize the high-performance, and so on.

5. Select proper architectural techniques for each product: For each product, select appropriate techniques, based on the policies determined in the previous section. We may not be able to select one architectural technique for each product using the basic policy. Typically we have to examine the followings to select a technique:

   - Compare candidates in terms of quality attributes. Even if multiple architectural candidates are applicable to the product, they do not have the same level of quality attribute. In the data retrieval example, A2 is better than A1 in terms of performance. Under the constraints of basic policy, select better candidate for the product. In this step, it is common to encounter the tradeoff problem. In the above example, A1 is better than A2 in terms of memory size. Therefore, we have to make decision considering which quality attribute is important.
   - Examine issues specific to the product. For example, even if we decide one architectural technique, say A1 is better than A2, we may select A2 if it is a common architecture in the market. This kind of examination is quite important but difficult to systematically do. If we introduce this kind of examination too early, it becomes a big bias of architectural design, and we may

design architecture just based on our impression. Therefore, we believe that it is important to examine architecture systematically in prior to considering this kind of issues.

4.3.5 The Meaning of Category of Requirements

In architectural design, we examine multiple requirements, and design software architecture to fulfill these requirements.

In functional design, there are a few ways to design software architecture to be the platform for multiple requirements. One is to develop software architecture to have every required functions. Each software selectively use these functions. One another is to develop software architecture to have the common part of the requirements. Each software is developed on the platform adding the difference on it. Furthermore, we could develop software architecture to have abstract functions for requirements, and each software defines concrete functions utilizing these architecture.

In quality aspect, on the other hand, the techniques are not well studied. However, one way is to make software architecture to be able to fulfill the wider range of requirements on quality attributes. We can compare the width based on the category of requirements. For example, in Table 4.13, A0 can fulfill the requirements for P1 and P2, and A1 can fulfill that for P1, P2 and P3. Therefore, A1 can fulfill the wider range of requirements. (Figure 4.7)

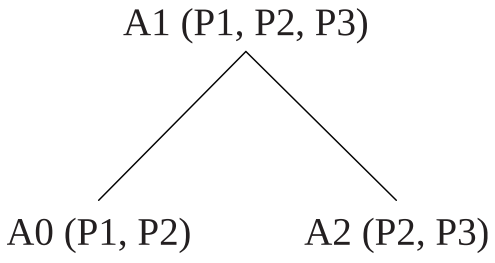


Figure 4.7: Partial Ordering based on the Category of Requirements

4.3.6 Impact of Infrastructure towards Quality Attributes

When we identify category of requirements and examine applicability, we do not consider the infrastructure. However, it is not rare that infrastructure of the software such as hardware, operating system and middleware would change during the evolution, or we have to develop products in product-line on multiple infrastructures. It is one of the typical techniques to adopt layered architecture in which we design a layer that hides the differences of infrastructures. However, this technique does not always work well, especially when we consider the quality attributes such as performance and memory-size.

Assume that we develop a family of information terminals that retrieve information stored in CD-ROM database. Though the CD-ROM format in the U.S. market is different from that in the Japanese market, we want to develop the same architecture for both markets. As the format of CD-ROM significantly influences the performance of the software because of its slow access speed, certain parts of the architecture have to depend on the CD-ROM format in order to fulfill required performance. Figure 4.8 shows the logical structure of the information terminal.

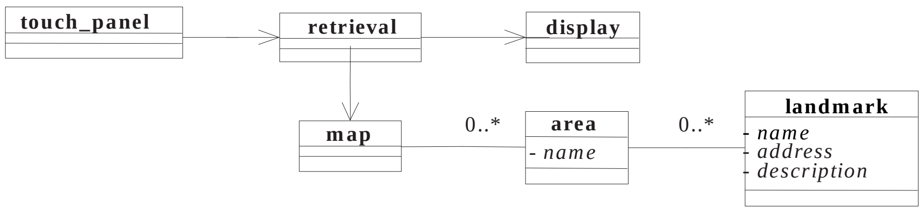


Figure 4.8: Infra Example: Logical Structure of Information Terminal

Consider the function that retrieves description that includes a given string s. As this function has to search every description text, physical format of CD-ROM has strong impact on the performance. Figure 4.9 and Figure 4.12 indicate physical formats of CD-ROM respectively. In these figures, the information about the physical format (physical arrangement) of the "description" data is shown. In format A, "description" data is grouped by the area in which corresponding landmark is included, and sorted by name. In format B, all the description data are placed in the continuous sectors, and sorted by id.

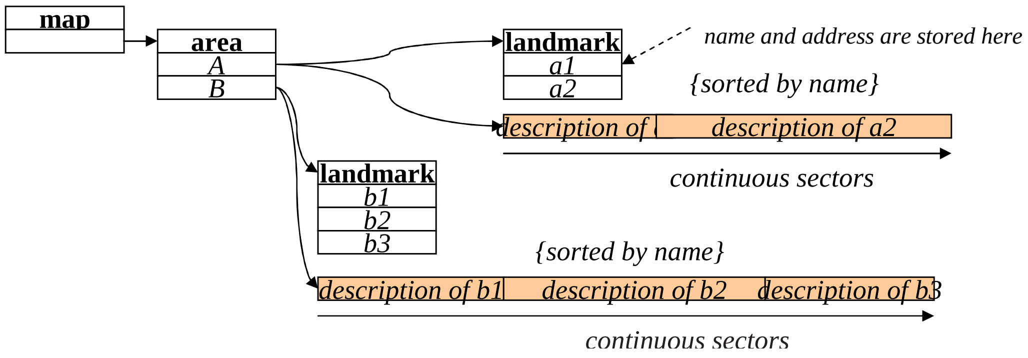


Figure 4.9: Infra Example: Physical Format of CD-ROM A

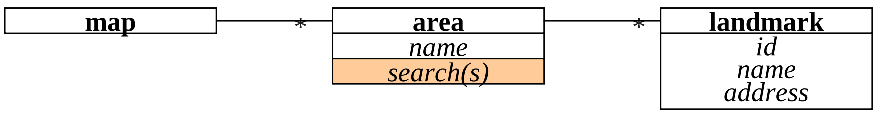


Figure 4.10: Infra Example: Proper Architecture for CD-ROM A

Based on the above analysis, we examined whether or not CD-ROM format has an impact on the architecture. To realize function that retrieves data on the CD-ROM format A is relatively straightforward. In the physical format, "description" data in the same "area" are placed in the continuous sectors, so we just search the description data from the beginning of the sectors. Therefore, we defined the method "search(s)" (search a string s from the set of "descriptions" included in the corresponding "area") in class "area" (Figure 4.10). On the other hand, to realize the function on the CD-ROM format

B is not so straightforward. In order to read the continuous sectors from the beginning and not to cause unnecessary seeks, we first retrieve the landmarks in the specified area, sort their id's, and read "description" data from the beginning of the sectors skipping unnecessary part. Therefore, we have defined the method "$get\_idlst()$" (get the list of id's included in the corresponding area) in class "area", and define the method "search(idlst, s)" (search a string s from all the "description" whose id is included in the idlist) in class "map" (Figure 4.12).

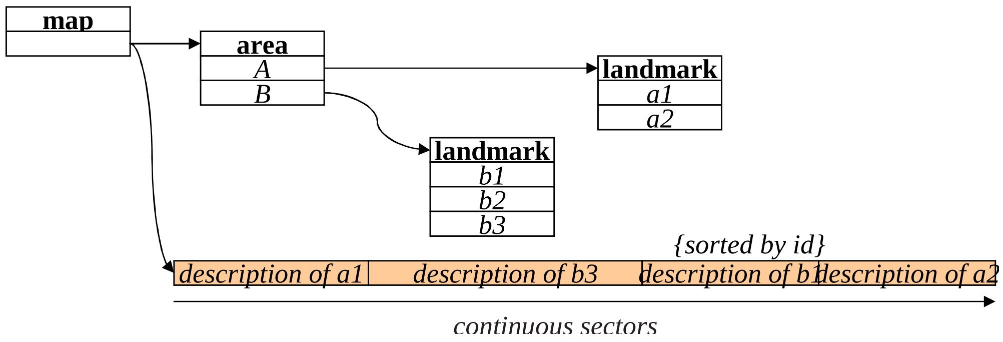


Figure 4.11: Infra Example: Physical Format of CD-ROM B

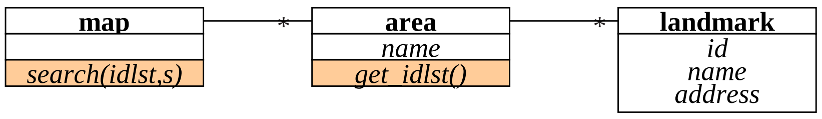


Figure 4.12: Infra Example: Proper Architecture for CD-ROM B

4.3.7 Category of Requirements and Infrastructure [27]

As we have examined in 4.3.6, when we have to consider multiple infrastructures, we have to carefully examine the impact of infrastructure towards quality attributes. As same as we examine requirements on potential software, we have to examine infrastructures of potential software. Assume that we have analyzed category of requirements to examine applicability.

There are three products P1, P2 and P3, and requirements on response time is shown in Table 4.14.

Table 4.14: Infra Example: Requirements on Products

	number	response time
P1	100	$<$ 2-3 sec
P2	1,000	$<$ 2-3 sec
P3	100,000	$<$ 2-3 sec

We have three architectural candidates A0, A1 and A2, and identified categories of requirements are shown in Table 4.15. Based on that we have decided the applicability

Table 4.15: Infra Example: Category of Requirements

	number	response	architectural techniques
CP1	100 - 1,000	$<$ 2-3 sec	A0, A1, A2
CP2	100,000	$<$ 2-3 sec	A1, A2

Table 4.16: Infra Example: Applicability Matrix

	P1	P2	P3
A0	x	x	
A1	x	x	x
A2	x	x	x

as Table 4.16. Here, we consider the infrastructure. Let P0 and P1 are developed on infrastructure I0, and P2 is developed on infrastructure I1. A0, A1 and A2 can fulfill performance specified as CP1 regardless of infrastructure, but in order to attain performance specified as CP2, A1 has to be developed on I0 and A2 has to be developed on I1.

Table 4.17: Infra Example: Category of Infrastructure

	number	response	candidates	infrastructure
CP1	100 - 1,000	$<$ 2-3 sec	A0, A1, A2	I0, I1
CP2	100,000	$<$ 2-d sec	A1	I0
			A2	I1

Category of infrastructure is defined for each category of requirements, and defined as a set of infrastructure on which same set of architectural techniques can fulfill the requirements included in the category of requirements (Table 4.17).

4.4 Decision-Making in Architectural Design [26, 29]

4.4.1 Architectural Design as Decision-Making Problem

As we have described in 3.2.1, we select architecture from architectural candidates that is most appropriate in terms of requirements on potential products developed on it and best matches the design policy. Though we can select architectures that satisfy the requirements on potential products using the technique based on category of requirements, there may remain multiple architectures that satisfy requirements. In such cases, we try to select one that best matches the design policy. We have mentioned in 3.2.1 that design policy gives the rules to determine the preference among candidates. In other words, selecting architecture that matches the design policy is to select most preferable architecture in terms of quality attributes. This process can be regarded as decision-making problem.

The other reason we are to apply decision-making framework toward architectural design is that we have to select architecture in the early phase of software development.

In the early phase, we do not have enough information about the architectural candidates, but we have to select design direction anyhow. We believe that when we select architecture based on information in hand, we need a systematic framework to do so, unless we select architecture just depending on our intuition. It is also important to make it possible to trace how we selected candidates. If we have leave information for tracing, we could check the validness of the decision, when we would obtain new information later on.

4.4.2 Decision-Making Framework

In decision-making, we select preferable one from multiple candidates in terms of decision criteria [20, 41]. In this paper, we consider architectural design as decision-making activity as follows (Figure 4.13):

- Candidates: We assume that architectural design is to select preferable architecture from architectural candidates in terms of decision criteria.
- Decision criteria: The decision criteria are quality attributes we are to consider when we design architecture. When we examine product-lines, we have to consider the quality of product-line as a whole, such as total development cost and reuse ratio. We will explain about quality of product-line in the next chapter.
- Preference: In general, we have to consider multiple criteria, and it is difficult to select single architecture that is superior to other candidates in terms of multiple criteria. We have to prioritize criteria so as to reflect the policy of product-line development.

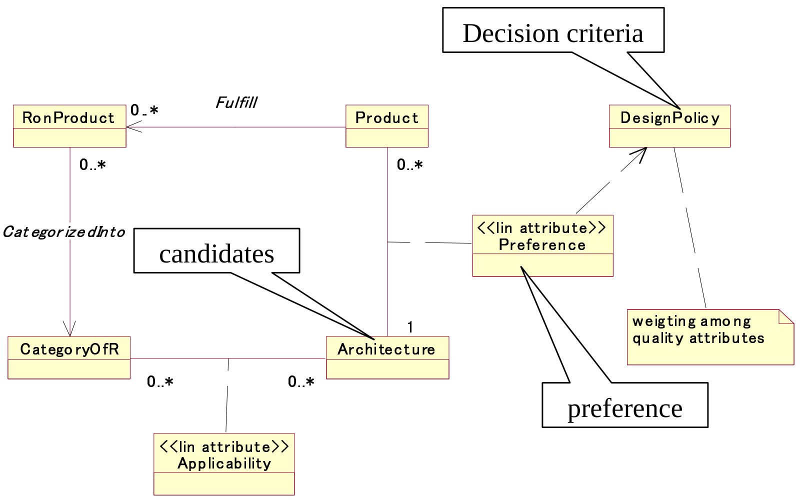


Figure 4.13: Decision Making Framework

4.4.3 Analytic Hierarchy Process (AHP)

Analytic Hierarchy Process (AHP) is one of the decision-making methods in which we select most preferable one from candidates in terms of design criteria [41]. In AHP, this scheme is depicted in Figure 4.14. Here, the target is to select most appropriate architecture, design criteria are seven quality attributes such as reliability, response and so on, and we have four candidates, option 1 to option 4. AHP is based on simple pair wise

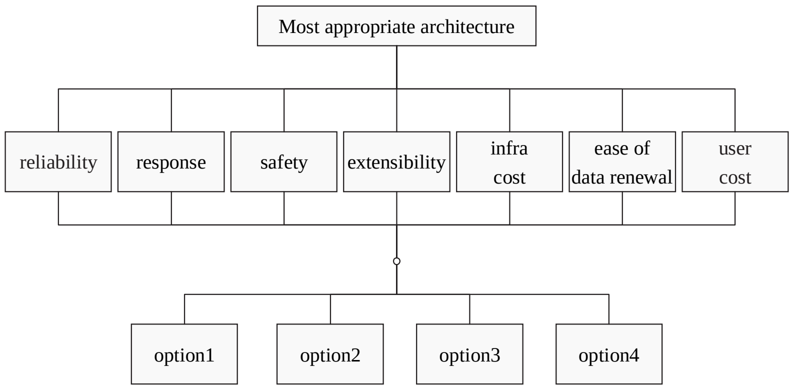


Figure 4.14: An Example of AHP Scheme

comparison among criteria and options. Firstly, we decide the relative importance among criteria based on pair wise comparison. Secondly, from each criterion, we also make pair wise comparison among architectural candidates. Finally, based on these comparisons, we can obtain overall priorities. We will explain the procedure in the next section.

4.4.4 Applying AHP to Architectural Design

Here, we have picked up an example from ITS on-board systems (see chapter 3.3 in detail) in which we select architecture for detecting vehicle position. As we have shown in Figure 4.14, we have seven quality attributes as criteria and four architectural candidates.

Firstly, we make pair wise comparison among criteria to decide the weight of each criterion. Table 4.18 shows the result of the pair wise comparison. Here, 1 means that two criteria are the same importance, 3 and 5 mean it is more important, and 1/3 and 1/5 mean it is less important. These values represent the relative importance. In the table, there represent that reliability is more important then safety but it is less important than user cost, and so on. Based on these pair wise comparisons, we can obtain weights of each criterion. We indicate the value in Table 4.18.

Secondly, for every criterion, we make pair-wise comparison among candidates. Table 4.19 shows pair wise comparison among candidates in terms of reliability. We also calculate the weight for every candidate. This table says that option2 is best in terms of reliability. We make this kind of comparison for other quality attributes.

When we got weights of criteria and weights of each option in terms of every criteria we can calculate the final weight for each option. Table 4.20 shows that option 4 is most preferable one among four options.

AHP is a tool to support decision-making and it gives us the weight among options based on pair wise comparison among criteria and options in terms of each criterion. Though it may be difficult to judge which option is the best, it is relatively easy and clear to judge small pair wise comparison. Based on these small decisions, AHP calculates the final weights among options in terms of design criteria. AHP shows just a consequence obtained by small decisions we made. Therefore AHP people warn that it is dangerous to

Table 4.18: AHP Example: Comparison among Criteria and Obtained Weights

	Relia-bility	Re-sponse	Safety	Extensi-bility	Infra. cost	Ease of data renewal	User cost	**weight**
Reliability	1	1	3	1	1	1/3	1/3	0.101
Response		1	3	1	1	1/3	1/3	0.101
Safety			1	1/3	1/3	1/5	1/5	0.041
Extensibiity				1	1	1/3	1/3	0.101
Infra. cost					1	1/3	1/3	0.101
Ease of data renewal						1	1	0.279
User cost							1	0.179

Table 4.19: AHP Example: Comparison among Options and Obtained Weights

	Option1	Option2	Option3	Option4	**weight**
Option1	1	1/7	1/5	1/5	0.052
Option2		1	3	3	0.528
Option3			1	1	0.21
Option4				1	0.21

Table 4.20: AHP Example: Final Weights of Options

	weight
Option1	0.18
Option2	0.3
Option3	0.205
Option4	0.319

use the weights without examining their appropriateness. They suggest that once we get weights, we should examine the result if it reflects our preference, and repeat the process until we get satisfactory weights.

In the case of making decision of architectural selection, we find that it is useful to check the portfolio of each options in terms of quality attributes, category of quality attributes (run-time, development-time, and so on) and to whom the quality gives benefit.

Figure 4.15 is a chart that shows portfolio of each candidate in terms of quality attributes. We understand that option4 is good for user cost and option2 is good for ease of data renewal. As we think that user cost is most important, we prefer option4. This chart visually shows characteristics of each candidate, and helps us to understand the meaning of weights obtained by AHP.

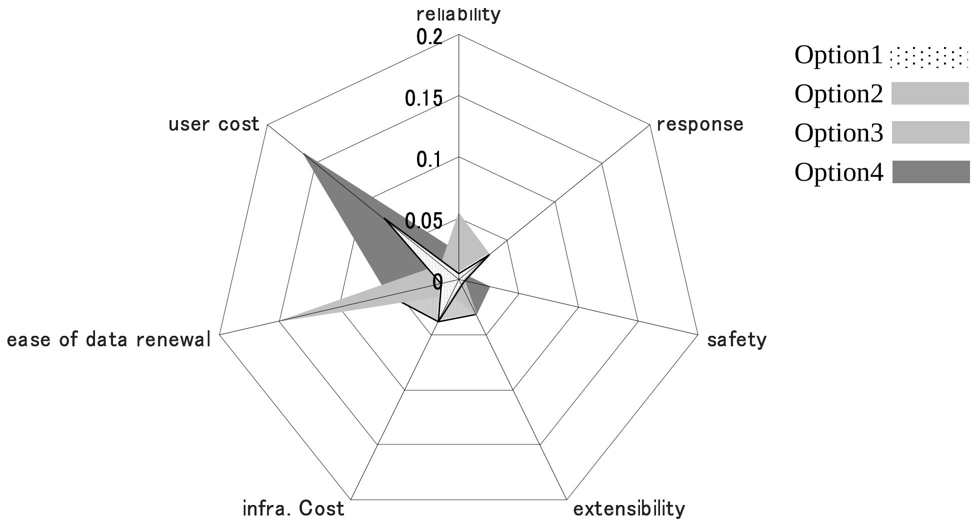


Figure 4.15: AHP Example: Portfolio (by Quality Attributes)

Figure 4.16 is another view of portfolio. As quality attributes can be categorized by means of the software structure, we categorized them into run-time quality attribute, operation-time quality attributes and development-time quality attributes. This shows that our preference is mainly based on operation-time quality attributes. If we think that run-time quality attributes or development quality attributes are more important than operation-time quality attributes, the preference may change.

Figure 4.17 shows the portfolio in terms of stakeholders, i.e. to whom these options are beneficial. As it is difficult to distinguish developers from operators (who manages the system), we categorize stakeholders into two, user and developer/operator. We understand option4 is good for user, and option2 is goof for developer/operator. This portfolio gives us the different view from those of other portfolios. Though both user cost and ease of data renewal are operation time quality attributes, the former is for user and the latter is for system manager.

Figure 4.18 shows the framework how we apply AHP to architectural selection.

- Firstly, we decide the policy for pair-wise comparison based on the design policy. For example, we decide the guideline of using the scale (such as 3, 1, 1/3).

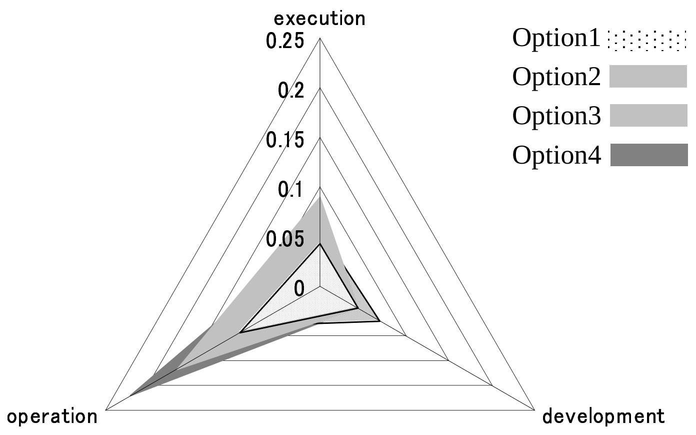


Figure 4.16: AHP Example: Portfolio (by Types of Quality Attributes)

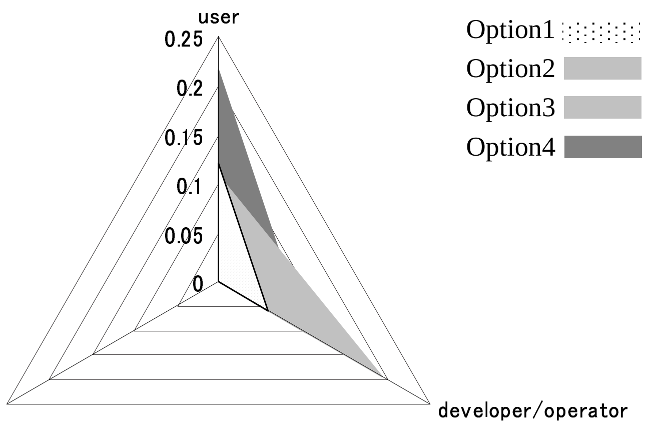


Figure 4.17: AHP Example: Portfolio (by Stakeholders)

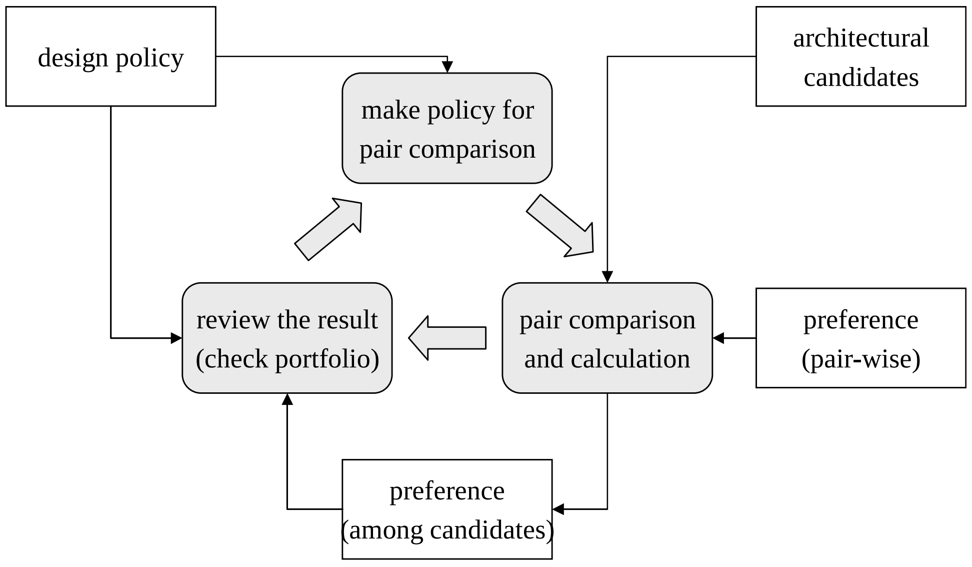

design policy
make policy for pair comparison
architectural candidates
review the result (check portfolio)
pair comparison and calculation
preference (pair-wise)
preference (among candidates)

Figure 4.18: Framework of Applying AHP to Architectural Design

- Secondly, we make pair-wise comparison among criteria and architectural candidates to calculate weights among them, and decide the preference among candidates in terms of design policy.
- Thirdly, we review the result. In this step, it is useful to analyze the portfolio of each candidate. We iterate this process until we will be satisfied with the result.

Through out these steps, it is assumed that we can decide the preferences for pair-wise comparison. AHP just shows us the preference among candidates, as the consequence of pair-wise comparison.

4.5 Determining Product-line Scoping [29]

4.5.1 Issues in Product-line Scoping

Product-line scoping is used to define the product-line. Namely it determines the products that comprise the product-line. Once a product-line is defined, we design a product-line architecture for it, and we examine the strategic development of the product-line utilizing the reusable assets based on the architecture. Therefore, when we define the scope, we have to examine whether or not it is appropriate for the products in product-line to share the architecture. In order to examine the appropriateness of sharing the architecture, it is useful not only to determine a set of products in the product-line, but also to examine what type of architecture would be shared by the determined products. In this paper, we define product-line scoping to be not only determining the member products but also examining architecture for the product-line. In other words, scoping is to divide a given set of products into one or more product-lines, and to examine the architecture for the product-line. Figure 4.19 shows an example of product-line scoping.

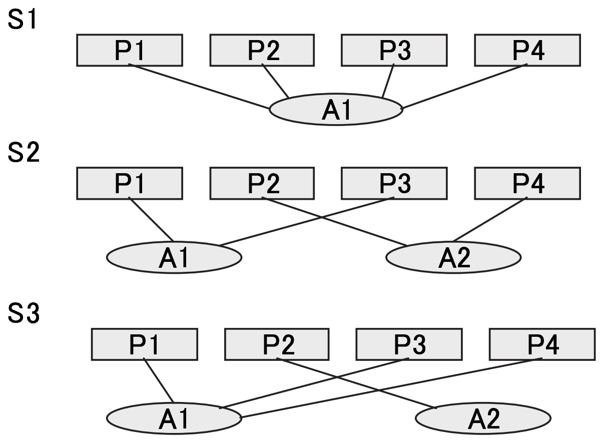


Figure 4.19: Example of Product-line Scoping

We describe a scope as a set of product-lines and a product-line as a pair of members-list and architecture. For example, assume that we have given a set of products P1, P2, P3 and P4, and A1 and A2 are architectural candidates for these products. Table 4.21 shows some examples of scope we could define:

Table 4.21: Example of Description of Product-line Scoping

S1	<P1, P2, P3, P4, A1>
S2	<P1, P3, A1>, <P2, P4, A2>
S3	<P1, P3, P4, A1>, <P2, A2>

S1 is a scope in which we define one product-line that includes every product and they share the architecture A1. S2 is a scope in which we define two product-lines, one includes P1 and P3, and the other includes P2 and P4, and so on. We do not prohibit a product-line to be consisting of a single product.

4.5.2 Requirements on Product and Product-lines

In our methods, we treat two types of requirements, one type is the requirements for each single product and the other type is the requirements for the product-line.

Requirements for single products are divided into requirements on functionality and requirements on quality attributes. In our method, we examine important services the product has to provide, and important requirements on quality attributes such as performance and memory size. As we examine the architecture for each product-line, we have to be careful whether or not the selected architecture is suitable for requirements for each single product. In this sense, the requirements for each product relate to individual optimal.

Requirements for product-lines show how we want to develop product-lines. Even if scoping is the same, the characteristic of product-line development may be different, for instance, development cost. For example, consider two products P1 and P2, which are developed sequentially, and we want to develop P2 faster, it is good to let P1 and P2 share the same architecture. Though there may be many complicated situations, we observe that there are typically two types of important situations in terms of determining the benefit of architecture sharing:

- Sequentially: The situation in which products are developed sequentially. For example, develop the high-end model first, and then develop the cost-down model.
- Co-existence: The situation in which products are developed simultaneously or in parallel. For example, develop products for different markets at the same time.

In the above cases, we can expect developing the products efficiently, if they share the same architecture. On the other hand, if we have to change the architecture, it would take time and the development becomes expensive. We do not claim that every product that is developed sequentially or in parallel has to share the architecture with other products. There may be a situation in which we want to release a product right after the previous product is released, or there may be another situation in which the quality of the product is more important than the release speed. Therefore, we should check the situation to determine the product-line scope. In other words, if we are not sure which products are developed first, or which products are developed simultaneously, we cannot judge whether they should share the architecture or not. In this paper, we describe sequential development and co-existence development as follow;

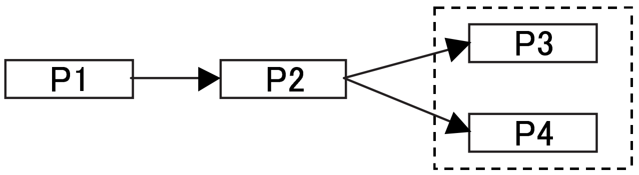


Figure 4.20: SCP Example: Description of Sequentially and Co-existence

In Figure 4.20, an arrow denotes sequential developments, and a dotted square denotes co-existent developments. For example, P2 is developed right after P1, and P3 and P4 are developed simultaneously after developing P2. When we examine requirements on product-lines, we firstly identify these sequential products, which are developed subsequently and which do co-exist. Next we evaluate whether a scope is good or bad in terms of the quality of the product-line. Consider the above example in Figure 4.20. There defined four sequential links and one co-existent link. When each requirement is as important as the others, we may define the scope so as to satisfy as many requirements as possible. If there defined priorities among them, we define the scope to satisfy the most important requirement. For example, assume that we want to develop P3 and P4 as fast as possible after developing P2. In such a case, Scope1 = { < {P1, P2, P3, P4}, A1 } and Scope2 = { < {P1}, A1 >, < {P2, P3, P4}, A2 > } are considered to be good in terms of this requirement, since P2, P3 and P4 share the same architecture. However, Scope3 = { < {P1, P2}, A1 >, < {P3, P4}, A2 > } is not considered good because in Scope3, the architecture for P3 and P4 are different from that of P2.

4.5.3 Design Policy

In determining the product-line scope, using requirements for each single product and for the product-line as a whole, we first have to judge whether or not the scope under consideration can fulfill the requirements. For example, assume that we have scope S = { < {P1, P2}, A1>, <{P3}, A2 > }. In order to fulfill the requirements on each product, A1 has to fulfill the requirements on P1 and P2; and A2 has to fulfill the requirements on P3.

Though we may restrict the number of scoping candidates by excluding candidates that do not satisfy the requirements, there may remain multiple candidates. In product-line scoping, we have to select one from these candidates. In such a case, we generally select the one that best matches the design policy of the product-line. Each candidate that satisfies the requirements has different characteristics. The design policy should give us the way to select from different candidates. In this paper, we characterize the candidates by means of quality attributes, and the design policy is defined as the priority among the quality attributes.

4.5.4 Applying Decision-Making Framework

During decision-making, we preferably select one from multiple candidates in terms of the decision criteria [20]. [41] We can apply this framework to product-line scoping, as this requires many decisions. For example, as we mentioned above, we have to select the most appropriate scope from multiple candidates of the product-line scope that satisfy different quality attributes. There is no objective way to decide which one is the best in terms of multiple quality attributes. We have to make a decision that is most suitable for our design policy.

In this paper, we consider scoping as a decision-making activity as follows:

- Options: We assume that scoping is to preferably select one from multiple candidates of the product-line scope in terms of the decision criteria.
- Decision criteria: There are two types of decision criteria. One type relates to requirements for each single product in terms of quality attributes such as performance and reliability [26]. The other type relates to requirements for the product-lines in terms of 'qualities' of product-line as a whole, such as total development cost and reuse ratio.
- Preference: In general, we have to consider multiple criteria, and it is difficult to select a single scope that is superior to other scopes in terms of multiple criteria. We have to prioritize criteria so as to reflect the policy of product-line development.

Chapter 5

Design Method

5.1 Design Policy

5.1.1 Relationship among Three Types of Architectural Design

We have categorized architectural design into three types, architectural design, product-line architectural design and product-line scoping. We will explain the design procedure for each type of architectural design from the following sections. However, these three design procedures are not completely independent. They share the same techniques we have explained in the previous chapters, and roughly speaking, product-line architectural design includes architectural design, and product-line scoping includes product-line architectural design. .

5.1.2 Basic Design Procedure

Basic procedure for architectural design is as follows:

1. Define design policy and scope of design: Clarify the target product or product-family to which we are to design architecture. We also have to clarify what kind of quality attributes we are to consider and define design policy in terms of these quality attributes.

2. Clarify requirements: Clarify the requirements on target product or product-family.

3. Examine the applicability: For each architectural candidate, examine whether or not it can fulfill the requirements in terms of each quality attribute. In this step, we utilize AOA and the idea of category of requirements.

4. Examine the preference: Though there may be multiple candidates that are applicable to the products, they have different characteristics. We examine the preference among candidates in terms of design policy.

5. Examine the tradeoffs: In product-line scoping, we have to examine the tradeoff between individual optimal and whole optimal.

6. Decide the design: Based on the result of examination on applicability and preference, we decide the design, namely select most appropriate architecture from architectural candidates.

This basic procedure is summarized in Figure 5.1.

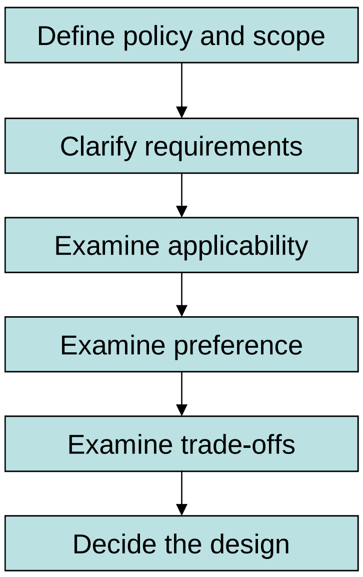


Figure 5.1: Basic Design Procedure

Our policy of defining design procedure is that even though we may not have enough information at architectural design phase, we examine current information systematically and analytically before making decisions, because it is quite important to make the design objective and traceable. We believe that it is important for architectural design not only to obtain good result but also to clarify the way how we reach the result, as architectural design is not a static process but dynamic process in which we check the appropriateness of decision and examine the effect to the result whenever new information about architectural design becomes available.

5.2 Simple Architectural Design for a Single Product

The followings are design procedure for architectural design for single product. Here we explain the procedure using a simple example.

1. Define design policy and scope of design: Assume the scenario of evolution or that of development of the product. Pick up target products to be analyzed from the scenario. We also decide quality attributes, from which we analyze requirements. For example, consider products that retrieve data in storage. Though our target is to design the architecture for P1, we assume evolution scenario for P1, and examine three products P1, P2 and P3 for the architectural design (Figure 5.2). We focus on the response time and memory size. We also define design policy. In which, we decide response time is more important than memory size. We think requirement on P1 is most important.

2. Clarify requirements: Table 5.1 shows the requirements on the quality attributes for these products.

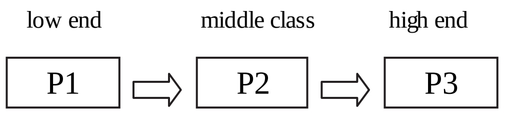


Figure 5.2: AD Example: Evolution Scenario

Table 5.1: AD Example: Requirements on Products

	number	response time	memory size
R1	100	< 2-3 sec	small
R2	1,00	< 2-3 sec	
R3	100,000	< 2-3 sec	

3. Examine the applicability: Identify the "category of requirements" for each quality attribute. Using the technique proposed in the previous section, identify categories of requirements for each quality attributes. In order to identify category of requirements, we pick up possible architectural candidates (Table 5.2).

Table 5.2: AD Example: Architectural Candidates

A0	Architecture that reflects the structure of analysis model.
A1	Architecture that adopts caching mechanism
A2	Load index data onto memory at the initialization time. It also has cache.

Table 5.3 and Table 5.4 show the categories of requirements we have identified for the data retrieval example. In these tables, '*' represents "no requirements". The categories of requirements from performance are as same as we have identified in the previous section. For memory size, we cannot use A2 if memory size is required to be small, because loading index data requires considerably large memory size.

Table 5.5 shows how to determine the applicability. From performance aspect, we can apply A0 to P1 and P2, and from size aspect, we can use A0 to every products. Therefore, if we are to fulfill both requirements, A0 is applicable to P1 and P2. Similarly, A1 is applicable to every product, and A2 is applicable to P2 and P3. Table 5.6 is a matrix obtained from the merged result. In this table, 'x' indicates "applicable".

Based on the analysis, we can understand the commonalities and differences of necessary architectural techniques. Table 5.6 shows what architectural techniques can be applicable to each product. P1 and P2 can use A0, but P3 cannot. Every product can use A1. P2 and P3 can use A2, but P1 cannot. Note that, applicable means that the requirements can be fulfilled, but it does not mean that every applicable architecture can attain the same level of quality attribute. For example, both A0 and A1 is applicable to P1, we can expect that A1 attains better performance than A0, and so on.

Table 5.3: AD Example: Category of Requirements (Performance)

	number	response	architectural techniques
CP1	100-1,000	< 2-3 sec	A0, A1, A2
CP2	100,000	< 2-3 sec	A1, A2

Table 5.4: AD Example: Category of Requirements (Size)

	memory size	architectural techniques
CS1	small	A0, A1
CS2	*	A0, A1, A2

Table 5.5: AD Example: Analyzing Commonality and Differences

	A0	A1	A2
performance	P1, P2	P1, P2, P3	P1, P2, P3
size	P1, P2, P3	P1, P2, P3	P2, P3
merged result	P1, P2	P1, P2, P3	P2, P3

Table 5.6: AD Example: Applicability Matrix

	P1	P2	P3
A0	x	x	
A1	x	x	x
A2		x	x

4. Examine the preference: Decide the preference among candidates using decision-making technique. Here we apply AHP to the problem in which we have two criteria performance and size, and three candidates A1, A2 and A3. We firstly make pair-wise comparison among criteria based on our design policy. Our design policy says that performance is more important than size, so we reflect the design policy to the comparison. Then, we make pair-wise comparison among candidates in terms of criteria. Table 5.7 shows the weights obtained by AHP.

Table 5.7: AD Example: Weights Obtained by AHP

	P1
A0	0.367
A1	0.406
A2	0.228

5. Decide the design: As design policy says that requirement on P1 is most important, we cannot select A2, because A2 does not satisfy requirements on size. On the other hand, we could select A0 because requirement on P3 is not so important as that on P1. However, considering the preference, A1 is better than A0. Therefore, we select A1 for our product.

5.3 Product-line Architectural Design

The followings are design procedure for architectural design for product-line.

1. Define design policy and scope of design: Identify products in product-line, and quality attributes to be considered. In this example, we consider P1, P2, P3 and P4 (Table 5.8), and consider response time and memory size. We also define design policy. In this example, we decide that memory size is most important for low-end model, and performance is most important for other products. We think P2 is most important product.

Table 5.8: PLA Example: Products in Product-line

P1	Low-end model for Japanese market.
P2	Standard model for Japanese market
P3	High-end model for Japanese. market
P4	High-end model for U.S. market

2. Clarify requirements: Identify the requirements on each product. Requirements on each product are defined in terms of quality attributes of products. Table 5.9 shows requirements on each product. This shows the requirements related to quality attributes of data retrieval, number of data in search space, response time and required memory size. As P1 is a low-end model, it has small number of data and is equipped with small memory. On the other hand, P3 and P4 have a large number of data, and so on.

Table 5.9: PLA Example: Requirements on Products

	number	response time	memory size
P1	100	< 2-3 sec	small
P2	1,000	< 2-3 sec	
P3	100,000	< 2-3 sec	
P4	100,000	< 2-3 sec	

3. Examine the applicability: Make a list of the candidate of architecture for given products. In this case, we make a list of candidates same as architectural design example (Table 5.2). In the previous step, we determine relative preference. In this step, we determine whether each candidate can fulfill the requirements. Firstly, assess each architectural candidate in terms of each quality attributes, and categorize requirements so as to each category of requirements can be fulfilled by the same set of architectural candidates (Table 5.10, Table 5.11).

Table 5.10: PLA Example: Category of Requirements (Performance)

	number	response	architectural techniques
CP1	100-1,000	< 2-3 sec	A0, A1, A2
CP2	100,000	< 2-3 sec	A1, A2

Table 5.11: PLA Example: Category of Requirements (Size)

	memory size	architectural techniques
CS1	small	A0, A1
CS"	*	A0, A1, A2

Secondly, examine requirements on each product and identify categories of requirements. For example, from performance aspect, A0 can be used for P1 and P2. From size aspect, A0 can be used for every product. So, we determine that A0 can be used for P1 and P2, because both performance and size have to be fulfilled (Table 5.12).

The Table 5.13 is another view of the result obtained by above examination. In this table, 'x' means "applicable"and blank means "non-applicable" (Table 5.13).

4. Examine the preference: For every product in product-line, examine the preference using AHP. Table 5.14 shows the result.

5. Decide the design: In this example, A1 is the only architecture that can be used by every product. Therefore, we select A1 for architecture for the product-line. However, note that A2 is most preferable architecture for P2, P3 and P4. In actual design, this may be a problem. In the next section, we examine the problem further.

Table 5.12: PLA Example: Examine Applicability

	A0	A1	A2
performance	P1, P2	P1, P2, P3, P4	P1, P2, P3, P4
size	P1, P2, P3, P4	P1, P2, P3, P4	P2, P3, P4
merged result	P1, P2	P1, P2, P3, P4	P2, P3, P4

Table 5.13: PLA Example: Applicability Matrix

	P1	P2	P3	P4
A0	x	x		
A1	x	x	x	x
A2		x	x	x

5.4 Product-line Scoping

In this section, we show the procedure to decide the scope of the product-line using a simple example. The example is a development of information terminal. Assume that we are to develop the following four products (Table 5.15).

The overview of our method is as follows:

- We first analyze the requirements for each product, and make a list of architectural candidates for each product. Then we evaluate these candidates from two aspects: One is to examine relative preference, and the other is to evaluate the applicability. This corresponds to examining product-line scoping from the view of the individual optimal.
- Then, based on this result, we make a list of the candidates of the product-line scope. We evaluate the requirements on the product-lines, and examine these candidates. This corresponds to examining product-line scoping from the view of the whole optimal.
- Finally, we examine the preference of scopes and the preference of each product, and determine the best scope, considering the quality of the product-line, and the quality of each product. In this step, we have to consider the tradeoffs between the individual optimal and the whole optimal.
- In the above steps, the relative preference is determined using a decision-making method; in our case we use the decision-making method AHP (Analytic Hierarchy Process [41]).

Table 5.14: PLA Example: Preference of Architectural Candidates

	P1	P2	P3	P4
A0	0.367	0.26	0.193	0.193
A1	0.406	0.357	0.307	0.307
A2	0.228	0.364	0.501	0.501

Table 5.15: PLS Example: Given Set of Products

P1	Low-end model for the Japanese market.
P2	Standard model for the Japanese market.
P3	High-end model for the Japanese market.
P4	High-end model for the U.S. market.

The following is the procedure of determining product-line scoping.

1. Identify the requirements: Identify the requirements for each product and the requirements for the product-line. The requirements for each product are defined in terms of the quality attributes of the products. The requirements for the product-line are defined in terms of sequentially and co-existing products.

   Table 5.16 shows requirements for each product. This shows the requirements related to the following quality attributes of data retrieval: the number of data elements in the search space, the response time and the required memory size. As P1 is a low-end model, it has small number of data elements and is equipped with a small memory. On the other hand, P3 and P4 have a large number of data elements, and so on.

   Table 5.16: PLS Example: Requirements on Products

	number	response time	memory size
P1	100	$<$ 2-3 sec	small
P2	1,000	$<$ 2-3 sec	
P3	100,000	$<$ 2-3 sec	
P4	100,000	$<$ 2-3 sec	

   Figure 5.3 shows the relations among products depicted by the notation we have introduced in the previous chapter.

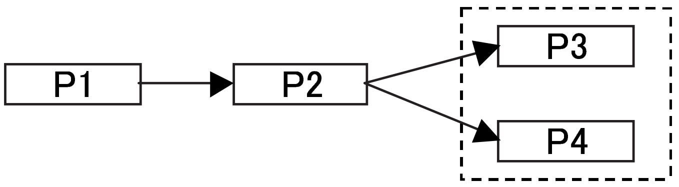
   

   Figure 5.3: PLS Example: Relations among Products

   Table 5.17 shows the requirements on the product-line. As we have shown in Figure 5.3, we develop P1 first, then develop P2, and finally develop P3 and P4 in parallel. Among these requirements, developing P3 and P4 in parallel is the most critical factor for developments of the products.

2. Define design policy: Define the design policy in terms of priority among the requirements. The following is the design policy for this example:

Table 5.17: PLS Example: Requirements on Product-line

Sequentially	P1, P2	
	P2, P3	More important than sequentially between P1 and P2
	P2, P4	More important than sequentially between P1 and P2.
Co-Existence	P3, P4	Most critical requirement for development of the product-line

- For the low-end model, reduction of the memory size is the most important.
- For the high-end model, the performance is the most important.
- If we can fulfill the above requirements, we would like to maximize the quality of each product.

3. Enumerate architectural candidate: Make a list of the candidate of architecture for the given products. For the data retrieval example, if the requirements on the quality attributes are not severe, we can fulfill them without considering any architectural technique. In such a case, we would design the architecture so as to reflect the structure of the analysis model that captures the nature of target domain, because this is considered to be good for the extensibility. However, if the requirements on quality attributes are severe, we may have to apply some architectural techniques to improve the quality attributes. In this example we have three candidates; A0 does not have any architectural technique to improve quality attributes, A1 has caching mechanism, and A2 loads index data of the storage onto memory at the initialization time (Table 5.18).

Table 5.18: PLS Example: Architectural Candidates

A0	Architecture that reflects the structure of analysis model.
A1	Architecture that adopts caching mechanism.
A2	Load index data onto memory at the initialization time. It also has cache.

4. Determine the preference of architectural candidate: In this step, we examine each candidate, and decide the relative preference among the applicable architectural candidates.

   We have applied AHP (Analytic Hierarchy Process) to decide the preference. AHP is one of the decision-making methods in which the most preferable one is selected in terms of decision criteria. In this case, the alternatives are the architectural candidates listed above, and the criteria are the quality attributes on which requirements are imposed, such as performance and reliability. AHP is based on simple pair-wise comparison among criteria and alternatives. First, we decide the relative importance among the criteria based on a pair-wise comparison. Second, for each criterion, we also make pair-wise comparison among the architectural candidates.

Table 5.19 shows the preference of the candidates for each product obtained by AHP.

Table 5.19: PLS Example: Preference of Architectural Candidates

	P1	P2	P3	P4
A0	0.367	0.26	0.193	0.193
A1	0.406	0.357	0.307	0.307
A2	0.228	0.364	0.501	0.501

5. Examine applicability of architectural candidate for each product: In this step, we determine whether each candidate can fulfill the requirements. First, assess each architectural candidate in terms of each quality attribute, and categorize requirements in such a way that each category of requirements can be fulfilled by the same set of architectural candidates. For example, we can fulfill the performance requirements by any architectural candidate if the number of data element is 100 to 1000, but only A1 and A2 can fulfill the performance requirement for 100,000 data elements. This leads to two types of requirements. Similarly we examine from size aspect, and find two types of requirements. Here, '*' denotes, no requirement (Table 5.20, Table 5.21).

Table 5.20: PLS Example: Category of Requirements (Performance)

	number	response	architectural techniques
CP1	100-1,000	$<$ 2-3 sec	A0, A1, A2
CP2	100,000	$<$ 2-3 sec	A1, A2

Table 5.21: PLS Example: Category of Requirements (Size)

	memory size	architectural techniques
CS1	small	A0, A1
CS2	*	A0, A1, A2

Second, examine the requirements for each product within the categories we found above. Determine the applicability of each architectural candidate to the products. For example, for the performance aspect, A0 can be used for P1 and P2. For the size aspect, A0 can be used for every product. Thus, we determine that A0 can be used for P1 and P2, because both performance and size requirements can be fulfilled (Table 5.22).

The Table 5.23 is another view of the result obtained by the above examination. In this table, 'x' means "applicable" and blank means "non-applicable".

6. Examine candidates for the product-line scope: (Examine candidates for the product-line scope.) We have to avoid assigning "non-applicable" architectural candidates to the products. For the data retrieval example, we prepare the following candidates

Table 5.22: PLS Example: Examine Applicability

	A0	A1	A2
performance	P1, P2	P1, P2, P3, P4	P1, P2, P3, P4
size	P1, P2, P3 ,P4	P1, P2, P3, P4	P2, P3, P4
merged result	P1, P2	P1, P2, P3, P4	P2, P3, P4

Table 5.23: PLS Example: Applicability Matrix

	P1	P2	P3	P4
A0	x	x		
A1	x	x	x	x
A2		x	x	x

(Table 5.24). Note that we do not intend to exhaustively enumerate the possible combinations. S1 is based on the idea to maximize the architectural sharing. S2 is to develop the low-end model on a different architecture than that the other models are developed on. S3 is to develop the high-end model on a different architecture than that the other models are developed on (Table 5.24).

Table 5.24: PLS Example: Candidate of Scope

S1	$<$ P1, P2, P3, P4, A1$>$
S2	$<$ P1, A1 $>$, $<$ P2, P3, P4, A2 $>$
S3	$<$ P1, P2, A1 $>$, $<$ P3, P4, A2 $>$

7. Determine preference among the candidates of the product-line scope: We determine the preference of the candidates of the product-line scope. Here, we also use AHP: candidates for the product-line scope are the alternatives, sequentially and co-existence are the decision-criteria. For sequentially, S1 is the best, as there is no architectural change during the development sequence, S2 is better than S3, because the requirements say that the sequence between P2 and P3, and the sequence between P2 and P4 are important, and S3 requires architectural change between them. For co-existence, S1 is also the best. S2 and S3 is the same, as the requirements say that co-existence of P3 and P4 is important, and in both scopes P3 and P4 share the same architecture.

   Based on these judgements, we have determined the following weight using AHP. We also obtain the sum of weights of each product defined in Table 5.19. For example, as S2 adopts A1 for P1, and A2 for P2, P3, P4, we get $0.406 + 0.364 + 0.501 + 0.501 = 1.772$. This weight represents the relative preference in terms of individual optimal (Table 5.25).

8. Define scope: Based on the examination so far, we determine the preferable scopes. As every products in S1 shares a single architecture, S1 is the best from the point of view of the whole optimal. However, sharing a single architecture does not maximize the quality of each product.

Table 5.25: PLS Example: Weight of Whole Optimal and Individual Optimal

	Whole optimal	Individual optimal
S1	0.619	1.377
S2	0.229	1.772
S3	0.153	1.765

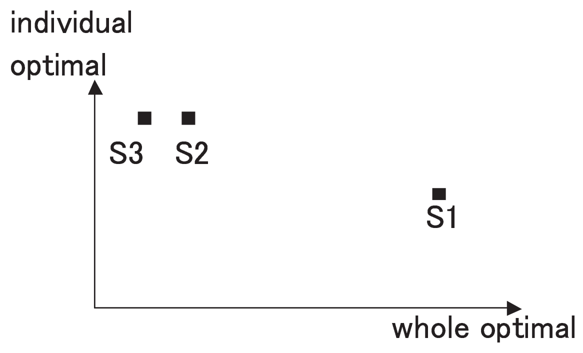


Figure 5.4: PLS Example: Portfolio of Each Scope

Figure 5.4 shows the situation. Note that, though the scopes have different characteristics, all of them satisfy the requirements on each product and the requirements on the product-lines. Based on the design policy, we want to maximize the individual optimal as far as we can fulfill the requirements on the product-line; we select S2 as our scope for this example.

Chapter 6

Case Study

6.1 ITS On-board System as Product-line Architectural Design

In this section, we actually apply the technique to ITS case, to demonstrate the applicability of the technique to actual problem.

The followings outline how we apply the technique.

1. Define design policy and scope of design: We pick up function C1-03 for this trial, because it is one of the most complicated cases, and used by 20 sub-services. We have applied the technique just using the knowledge about the quality attributes written in the report, and not using other factors. C1-03 is a function that detects the position of vehicle. For this function, they have requirements on the following four quality attributes: accuracy, road coverage, vehicle coverage, and cost. We summarize the meaning of these quality attributes in Table 6.1.

   Table 6.1: Casestudy 1: Quality Attributes to be Examined

accuracy	The accuracy of detected position.
road coverage	The ratio of the area (road) where this function works. If vehicle has the function, it may work everywhere, but if some equipment is required on roadside, it may be difficult to cover the entire road, because we have to equip them everywhere.
vehicle coverage	The ratio of the vehicle to which this function works. If vehicle has to have some equipment, it may be difficult to detect every vehicle because we have to equip them to every vehicle.
cost	Cost of developing infrastructure on roadside.

   There are 20 sub-services that use the function, and we have examined these sub-service in the following steps.

   We have defined two design policies. Firstly, we try to maximize the sharing of architectural technique. Secondly we minimize the cost if possible.

2. Clarify requirements: Table 6.2 and Table 6.3 summarizes the requirements on these functions.

Though the decision where the function has to be placed is not specified in analysis model, they have decided the global architectural style for some sub-services and these decisions determine the placement of the functions. For example, if they decide to make the service without any support of roadside equipment, we have to place the function on vehicle, and so on. Based on this global architecture, we have divided sub-services into two; one is for vehicle, and the other for roadside. Some sub-services require placing the function both side.

The number of each sub-service is the id used in the projects. These requirements are picked up from the report, and blank columns mean there are no written requirements in the report.

Table 6.2: Casestudy 1: Requirements on Each Sub Services (Vehicle)

	accuracy	road coverage	vehicle coverage	cost
01	high	everywhere		
02	high	everywhere		
03	high	everywhere		
04	low	everywhere		low
12				
14				
15				
16				
19	high			
28	low	everywhere		low
29	low	everywhere		low
32	low	everywhere		low
33	low	everywhere		low
40				
41				
42				
43				
45				

3. Examine the applicability: Table 6.4 shows the list of architectural candidate. (a), (b) and (c) is for detecting on vehicle, and (d), (e) and (f) is detecting on roadside.

   Table 6.5 and Table 6.6 show the categories of requirements for each quality attribute.

   Table 6.7, Table 6.8, Table 6.9, and Table 6.10 show the result of analyzing commonalities and differences.

4. Examine the preference: Though we could use AHP to determine the preference, in this case preference is easily judged. For vehicle, we prefer (b) if accuracy is required, unless we prefer (a) because it is good for cost. For road-side, we prefer (f), if applicable, as it is good for cost.

Table 6.3: Casestudy 1: Requirements on Each Sub Services (Roadside)

	accuracy	road coverage	vehicle coverage	cost
12			every vehicle	
14				
15				
16				
27		urban area		low
31		specific route only		
34		specific route only		
41	high			

Table 6.4: Casestudy 1: Architectural Candidates for C1-03

(a)	Detecting position only by vehicle (no communication with roadside or center).
(b)	Detecting position by vehicle, and use the data from roadside and center to correct the error. (DGPS and KGPS are included in this.)
(c)	Detecting position using the data from roadside, then reflect the data of database or gyro in vehicle.
(d)	Collecting data obtained by (a) or (b) using inter-center communication.
(e)	Detecting the vehicle from roadside equipment by receiving information transmitted from vehicle.
(f)	Detecting the vehicle from roadside equipment by reading the license number, and so on.

Table 6.5: Casestudy 1: Categories of Requirements (Vehicle)

accuracy	candidate
high	(b)
low/*	(a)(b)(c)

road coverage	candidate
everywhere/*	(a)(b)(c)

vehicle coverage	candidate
*	(a)(b)(c)

cost	candidate
*	(a)(b)(c)

Table 6.6: Casestudy 1: Categories of Requirements (Roadside)

accuracy	candidate
high/*	(d)(e)(f)

road coverage	candidate
urban/road	(e)
*	(d)(e)(f)

vehicle coverage	candidate
every vehicle	(f)
*	(d)(e)(f)

cost	candidate
*	(d)(e)(f)

Table 6.7: Casestudy 1: Analyzing Commonalities and Differences (Vehicle)

	(a)	(b)	(c)
accuracy	other than 01, 02, 03, 19, 45	all	other than 01, 02, 03, 19, 45
road coverage	all	all	all
vehicle coverage	all	all	all
cost	all	all	all
merged result	other than 01, 02, 03, 19, 45	all	other than 01, 02, 03, 19, 45

Table 6.8: Casestudy 1: Analyzing Commonalities and Differences (Roadside)

	(d)	(e)	(f)
accuracy	all	all	all
road coverage	other than 27, 31, 34	all	other than 27, 31, 34
vehicle coverage	other than 12	other than 12	all
cost	all	all	all
merged result	other than 12, 27, 31, 34	other than 12	other than 27, 31, 34

Table 6.9: Casestudy 1: Applicability Matrix (Vehicle)

	01	02	03	04	12	14	15	16	19	28	29	32	33	40	41	42	43	45
(a)				x	x	x	x	x		x	x	x	x	x	x	x	x	
(b)	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
(c)				x	x	x	x	x		x	x	x	x	x	x	x	x	

Table 6.10: Casestudy 1: Applicability Matrix (Roadside)

	12	14	15	16	27	31	34	41
(d)		x	x	x				x
(e)		x	x	x	x	x	x	x
(f)	x	x	x	x				x

5. Decide the design: The followings are results of architectural selection based on the technique. For vehicle, if we have multiple candidates, we select (a) as it is good for cost. For roadside, if we have multiple candidates, we select (f) as it is good for cost. Table 6.11 and Table 6.12 show the result of architectural selection: "result" is our selection, and "original" is the result of the project.

Table 6.11: Casestudy 1: Result of Architecture Selection (Vehicle)

	01	02	03	04	12	14	15	16	19	28	29	32	33	40	41	42	43	45
result	b	b	b	a	a	a	a	a	b	a	a	a	a	a	a	a	a	b
original	**a** **b**	**a** **b**	**a** **b**	a	a b	a b	a b	a b	**a**	a b	a b	a b	a b	a	a	a	a	b

Table 6.12: Casestudy 1: Result of Architecture Selection (Roadside)

	12	14	15	16	27	31	34	41
result	f	f	f	f	e	e	e	f
original	f	f	f	f	**d,e**	e	e	f

The followings are some analysis about the result: We analyze the reason why we have different result for 19. The reason is that in the project, candidate (a) is selected because it is already used in the current system. As we do not use such information, the result becomes different. For some sub-services, they select more than one candidates. However, in most of the case, our selection is included in their candidates, and we do not think it is the major differences. For 01, 02, 03, we have omitted the (a) as it is not applicable (Table 6.9). The reason of these differences is that they select the candidate nevertheless they explicitly pointing out that this candidate has not enough accuracy. For 27, we also omit the (d) (Table 6.10), but it is selected as one of the candidate. They select (d) as it is applicable to urban area for some reason. Except these points, we have obtained the considerably reliable result using the technique without considering other factors.

This trial shows that we can apply the technique to actual problem, and we have obtained considerably reliable result by this systematic procedure.

6.2 ITS On-board System as Product-line Scoping

In this section, we pick up some products (sub-services) examined in the projects, and determine the scope using the method. We pick up the following products: P02, P04 and P19.

We have to develop each product in 2003 and 2008 (Table 6.13).

In this case study, we focus on the year 2003 and 2008. As the requirements on each product and technology may be different in these years, we distinguish the products in 2003 from those in 2008. We attach the year behind the product name to denote the year. For example, P02-2003 means product P02 in 2003. In this section, we focus on the architecture that realizes the function "detect the vehicle position", one of the most important functions and it has strong impact on the entire architecture for these products.

Table 6.13: Casestudy 2: Products

P02	Route guidance
P04	Provision of road traffic information
P09	Notification to emergency center

1. Identify the requirements: We consider the following quality attributes (Table 6.14).

Table 6.14: Casestudy 2: Quality Attributes to be Examined

accuracy	The accuracy of detected position.
road coverage	The ratio of the area (road) where this function works. If the vehicle has the function, it may work everywhere, but if some equipment is required on the roadside, it may be difficult to cover the entire road, because we have to equip them everywhere.
vehicle coverage	The ratio of the vehicle to which this function works. If the vehicle has to have some equipment, it may be difficult to detect every vehicle because we have to them to all vehicles.
cost	Cost of developing the infrastructure on the roadside.

The followings are the requirements for the products in 2003. For the 2008 version, the requirements are the same, except that the accuracy for P02 becomes higher (Table 6.15).

Table 6.15: Casestudy 2: Requirements on Products

	accuracy	road coverage	vehicle coverage	cost
P02	high	everywhere		
P04	low	everywhere		low
P19	high			

Figure 6.1 shows the relations among the products.

The requirements on the product-line are as follows (Table 6.16).

2. Define the design policy: The Following is the design policy.

- For P02, the accuracy is the most important. However, in 2003, the requirements are not so severe as we do not have enough services that require such accuracy.
- For P04, the cost is the most important.
- For P19, the accuracy is the most important, as this product relates to safety.

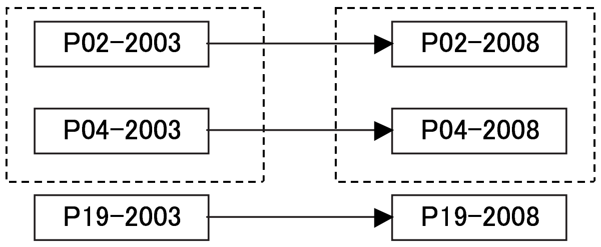


Figure 6.1: Casestudy 2: Relationship among Products

Table 6.16: Casestudy 2: Requirements on Product-lines

Sequentially	P02-2003, P02-2008	same importance
	P04-2003, P04-2008	
	P19-2003, P04-2008	
Co-Existence	P02-2003, P04-2003, P19-2003	same importance
	P02-2008, P04-2008, P19-2008	

- If the product can fulfill the above requirements, we would like to maximize the quality of each product.

3. Enumerate architectural candidate: Table 6.17 shows the architectural candidates for "detect vehicle position".

Table 6.17: Casestudy 2: Architectural Candidate

(a)	Detect the position by the vehicle only (no communication with the roadside or a center).
(b)	Detect the position by the vehicle, and use the data from the roadside and the center to correct the error. (DGPS and KGPS are included in this.)
(c)	Detect the position using the data from the roadside, next reflect the data of the database or the gyro in the vehicle.

4. Determine the preference of the architectural candidate: Using the accuracy, road coverage, vehicle coverage and cost as criteria, we determine the following preference using AHP (Table 6.18).

5. Examine the applicability of the architectural candidate for each product: Using the procedure we have discussed in the previous section, we determine the following applicability. There are no differences between 2003 and 2008 except for P02 (Table 6.19).

6. Examine the candidates for the product-line scopes: We have examined the following candidates for the product-line scopes (Table 6.20).

Table 6.18: Casestudy 2: Preference of Architectural Candidate

	P02-2003	P02-2008	P04-2003	P04-2008	P19-2003	P19-2008
(a)	0.364	0.282	0.481	0.481	0.286	0.286
(b)	0.414	0.482	0.314	0.314	0.485	0.485
(c)	0.222	0.227	0.205	0.205	0.229	0.229

Table 6.19: Casestudy 2: Applicability Matrix

	P02-2003	P02-2008	P04-2003	P04-2008	P19-2003	P19-2008
(a)	x		x	x		
(b)	x	x	x	x	x	x
(c)			x	x		

7. Determine the preference among the candidates for the product-line scope: The following shows the weights that represent the whole optimal and the individual optimal for each candidate of scope (Table 6.21).
8. Define scope: Based on the design policy, we select S2 as the scope (Figure 6.2).

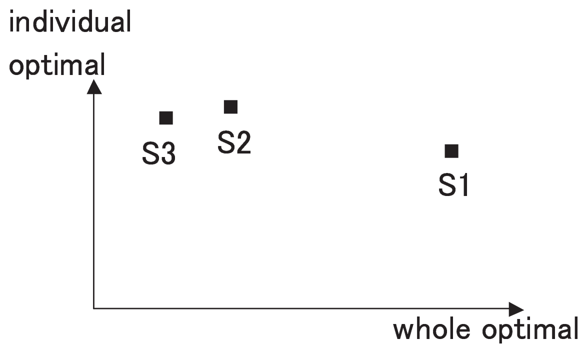


Figure 6.2: Casestudy 2: Portfolio of Scopes

This decision means that in 2003, we develop P02 and P04 on a relatively inaccurate and low-cost architecture, as these services are for comfortable driving, but do not directly relate to safety. But, for P19, we adopt an expensive but accurate technology as it relates to safety. In 2008, the functionality of each product is upgraded, and they require an architecture that is more accurate. The result matches our value.

Table 6.20: Casestudy 2: Candidate of Scope

S0	{<P02-2003, P02-2008, P04-2003, P04-2008, P19-2003, P19-2008, (b)>}
S1	{ < { P02-2003, P02-2008, P19-2003, P19-2008}, (b) >, < { P04-2003, P04-2008}, (a) > }
S2	{ < { P02-2003, P04-2003}, (a) > , < { P02-2008, P04-2008, P19-2003, P19-2008}, (b) > }

Table 6.21: Casestudy 2: Weight of the Whole Optimal and Individual Optimal

	Whole optimal	Individual optimal
S1	0.644	2.494
S2	0.24	2.828
S3	0.116	2.611

The case study demonstrates the following:

- We can determine the scope of the product-line by a systematic way, considering quality attributes of each product and quality of the product-line.
- We can determine the scope of a product-line in a systematic way, and we can obtain the result with enough resolution that is required in the actual projects.
- We can apply the method to the case from the actual project that was carried out by more than 10 experts, and we can obtain the result with enough resolution.

Chapter 7

Evaluation and Discussion

7.1 Evaluation of the Case Study

In 1.2, we have mentioned the conditions for practical methodology, size, cost and resolution. We evaluate the method in terms of these conditions.

7.1.1 Size

The case study is based on the actual project, and we have applied our technique to the problem using exactly the same procedure we have proposed in this paper, and have obtained the result. As we have introduced in 3.3.1, the project is large projects, i.e. it takes nearly two years just for architectural design. As the project is for investigating the direction of development of ITS on-board systems, we have to exhaustively enumerate possible alternatives and evaluate them to make the result reliable and acceptable for every stakeholder. We have extracted dozens of important functions, enumerated architectural candidates for each of them, and evaluated the appropriateness for each sub-services. In this sense, the size of the projects is quite large in terms of number of architectural candidates and number of evaluations.

Though the project is quite large, the focus of the project is to determine the design direction of ITS systems in the early phase of development, and does not include implementation issues. The architectural design for finding design directions, and that for actual implementation have different characteristics. We will discuss on the difference in 7.2.2.

7.1.2 Cost

We examine whether the cost required by the method is reasonable or not by comparing the cost with that of the actual ITS on-board architectural design. As it is difficult to compare the cost numerically, we examine the cost quantitatively. Table 7.1 summarizes the differences.

The table compares similar step of each method. Our method requires extra step to separate requirements on each quality attribute from original requirements. On the other hand, actual method requires characterizing each architectural candidate in terms of quality attribute. Though we cannot quantitatively argue the cost from this table,

Table 7.1: Quantitative Comparison of Cost

Actual ITS on-boar architecture design Method	Our method
	For every important requirement, separate requirements on quality attribute using AOA.
For every important requirements, list up architectural candidates.	For every quality attribute, list up architectural techniques considering the category of requirements.
For every architectural candidate, examine its characteristic in terms of quality attributes	
For every important requirement, examine the appropriateness of each architectural candidate.	For every important requirement, examine applicability and preference of each architectural candidate.

we can say that our method requires more atomic works and requires a little more cost. However, as far as we examine our case study, the difference is not so large.

In the actual project, we simply characterize each candidate in terms of quality attribute. On the other hand, in our method, we identify the category of requirements. Though identifying the category of requirements requires higher skills, this categorization becomes the template of examining the applicability, and helps us to decide applicability in the consistent way.

7.1.3 Resolution

As we have made the observation in 3.3.2 and the demonstration in 6.1, we can distinguish architectural candidates and select suitable one depending on the requirements and design policy as we have done in actual project. As ITS example is higher-level architectural design, one may claim that we need more fine-grained resolution if we are to apply the method lower-level architectural design. However, we believe that a basic framework for architectural evaluation, in which we compare possible architectural candidates in the objective way, should not be so complicated, because complicated methods are difficult to understand and expensive.

7.2 Applicability of the Method

7.2.1 Model for Software Structure

As we have explained in 3.4, our modeling framework of software structure is based on the ordinary object-oriented modeling framework, this framework can be applied to wide range of software structure, from high-level modeling like ITS case to low-level modeling like information terminal case. Though the way to describe requirements on quality attributes is extended by us, it is natural extension of UML. This is quite similar to existing documentation technique in which we describe requirements on quality attributes as textual form in documents that are structured by components or functions.

7.2.2 Granularity and Abstraction Level

As we have claimed in 7.2.1, we can model architecture at wide spectrum of abstraction level and granularity. However, the characteristics of resulting model become different. The major difference is relationship among modeled components and collaborations. In our ITS example, typical relationship among collaborations can be categorized into two:

- One is vertical relationship, that is higher level decision constrains lower level decisions. For example, if we decide at higher level to deploy heavy processing unit at center side, and make on-board system as light as possible, the lower architectural decisions have to follow the higher policy. We cannot adopt components or collaborations that violate the policy.
- The other is horizontal relationship, that is components and collaborations at the same level impose constraints each other. They have to share the same architectural assumption such as basic policy of data handling, control architecture, assignment of responsibility to each subsystems, unless they cannot work correctly.

Even though we design at low-level architectural design, relationship among components and collaborations can be typically categorized into these two. However, at low-level architectural design, the model becomes more concrete and finer, and the relationship also becomes complicated. Consequently, we have to carefully select architectural candidates considering these constraints.

7.2.3 Types of Quality Attributes

In 3.4.2, we have discussed that though it is possible to describe requirements on quality attributes in our modeling framework, we do not insist that we could consider requirements on every quality attribute by our method. As a matter of fact, for some quality attributes, we do not find strong and direct relationship with software structure. If factors that determine quality attributes are not naturally described on architectural model, we do not think we can consider the quality attributes using the techniques proposed in this paper. For example, though some aspects of usability may be determined by software structure, non-architectural issues such as human interface design have a stronger impact on the quality attribute.

7.3 Comparison with other Techniques

7.3.1 Evaluation Techniques

In selecting architecture from multiple options, we have to evaluate each option in terms of requirements on quality attributes. Therefore evaluation technique is one of the key issues in architectural design.

The followings are typical techniques for architecture evaluation;

- ADL: Describe software architecture by ADL and evaluate the characteristics of the architecture by mathematical analysis or simulation. This technique is rigorous, but is only applicable to some restricted field [1, 9, 34, 35, 43].

- Performance model: There are many researches on performance model in the performance engineering field, and there are researches on software architecture field that utilize the performance model [22]. Some of the ADL approaches above utilize this technique. There are also researches in which they map UML models onto performance model [16]. There also researches in which they link architectural style with the performance model [30].

- Scoring: Set up some scoring criteria, and evaluate each option in terms of the criteria, mainly by qualitative way. This method can be used easily and widely, and already used in the actual software field. However, in order to make the technique properly work, there still remain many issues such as how to set up criteria, how to make qualitative evaluation, how to decide the weight for each criterion, and so on. SAAM is one of the typical scoring techniques that utilize the scenario [21].

- Review: It is a common techniques in software development to review some artifacts, such as documents, source code and test cases. For architecture, there are typically two types of reviews, one is architecture review in the early phase of software development, and the other is that in the actual design phase. In this architectural review, the role of architect is quite important [32]. Among studies on architectural review, ATAM is the technique that utilize architectural catalogues to analyze characteristics of some quality attributes [22].

- Heuristics: There are researches in which they utilize heuristics commonly used in architectural design. RARE is one of the environment, in which they support software generation based on the decisions in terms of these heuristics [3].

Our method can be categorized into the scoring method. As our focus is architectural design in the early phase of software architectural design in which we evaluate architectures from spectrum of quality attributes. The scoring is one of the best approaches for such purpose. Furthermore, our experiences in the ITS project require to make the architectural design as objective as possible, because it is a national project and has to make the result open and accountable.

7.3.2 Scoring Techniques

There are many ways to evaluate architecture by scoring. The followings are typical criteria for scoring;

- Scenario: In SAAM, they pick up important scenarios, and score architectures in terms of these scenarios. Scenario reflects the actual use of the system and make the evaluation real. On the other hand, it is difficult to decide the good set of scenarios for architectural evaluation [21].

- Parameter/Key: In selecting architecture, there must be some important parameters or keys that affect the selection. There are researches in which they utilize these parameters and keys for architecture selection. This technique is good as it reflects the actual way of architecture selection, but tends to domain and application dependent, and it is difficult to find the general framework for architectural selection [3, 33].

- Quality Attributes: Using the quality attributes (or some sub characteristics) for architectural evaluation. This technique is good for make the scoring objective and reliable, but it is difficult to set up set of criteria and decide the weight of each criterion.

Our method utilizes the quality attributes as evaluation criteria, as it is good for making the scoring objective. The criteria mentioned above are not exclusive. For example, combining scenario-base evaluation with quality attributes based evaluation, may compliment each other.

7.3.3 Decision-Making in Architectural Design

Our decision-making technique is based on the decision-making framework proposed by Jozwiak [20]. They showed general and abstract framework of decision-making issues in software design. Our method is conceptually based on this framework, but concretely argues how to apply the framework to software architectural case.

In our method, we use the quality attributes as decision criteria. We can define different software architecture on different infrastructure; such as run-time software architecture and development-time software architecture. Each architecture relates to corresponding quality attribute; for example, run-time architecture determines run-time quality attributes. Therefore, using these quality attributes as decision criteria makes it easy to understand the relationship among the decision and corresponding architectural design.

We have used the technique to decide the preference among the applicable candidates. Once we have chosen applicable options, we want to select one that best matches the design policy. As we have defined design policy by means of weighting among quality attributes, namely decision criteria, we can determine the preference consistent with design policy. We have also applied decision-making technique to examine the whole-optimal and individual-optimal; we analytically evaluate architecture in terms of quality attributes, and based on that determine these optimal, using decision-making technique.

Among many decision-making techniques, we have adopted AHP [41]. This technique is suitable for architectural design, as it is easy to use; we can determine the preference based on the simple pair-wise comparisons. In early phase of architectural design, we do not have enough information to make comparisons. AHP can be used in such a case; we can calculate the weight, even if we leave some uncertain pair-wise comparison undecided.

7.3.4 Examples of Existing Techniques

The followings are comparisons with famous techniques.

SAAM [21]

As our method supports evaluating design alternatives to select most appropriate architecture from alternatives, evaluation is one of the important techniques. One of the famous evaluation techniques for architectural design is Software Architecture Analysis Method (SAAM). This method is to evaluate multiple architectural candidates by scoring utilizing scenarios. One of the typical examples of utilizing SAAM is to evaluate extensibility of the software by examining what part of architecture requires change if the extension specified in scenario would occur.

Comparing our method that gives general framework, SAAM is good for evaluating architecture in terms of specific quality attributes such as extensibility. On the other hand, our method gives general framework for evaluation for many quality attributes.

ATAM [22]

The Architecture Tradeoff Analysis Method (ATAM) is a method for architecture evaluation. It provides a set of data for us to understand the architecture and to judge whether the architecture can meet the requirements on quality attributes or not. It means that we do not get any numerical value of the architecture, which automatically leads us to a final decision, by the method, but we can understand and consider the architecture based on the data obtained by the method. In that sense, it resembles the architecture review approach. The data obtained by the method include which parts of the architecture are critical for a quality attribute, and which parts of the architecture are related to tradeoff problems.

The method utilizes scenarios greatly. They analyze architectural approaches (the set of architectural decisions) used in the architecture based on the scenarios. The architectural approaches can be described using architectural styles. Especially, the Attribute Based Architectural Styles (ABAS) [30] are very useful in the method. We could utilize this kind of review technique in our method, when we examine the characteristics of architectural candidates.

Chapter 8

Conclusion

We have proposed an architectural design method in which we consider requirements on quality attributes, and support decision-making we encounter during architectural design.

This research was motivated by the ITS on-board system design, we introduced in this paper, in which we have to objectively design higher architecture. As the system is large, complicated, and socially important, we have to adopt a systematic way for the architectural design. We have adopted scoring technique in actual design. After completing the projet, we observerd the process to see what we did actually and how it worked. From this observation, we found out it is important to analytically examine quality attributes, to identify category of requirements and to handle decision-making issues.

In actual software development, it has not been so popular to adopt a systematic method for architectural design. The actual practice in daily software development projects is that a few skilled software designers gather and discuss the design direction mostly based on their knowledge, experience and intuition. In the larger projects, on the other hand, they spend more time on discussing architectural design based on the comparison of possible design alternatives; sometimes they make prototypes to validate the design or compare the alternatives. However they lack a systematic method to determine software design direction.

However, recent demands on software such as continuous evolution, spiral development, and strategic development of product-lines make software architecture more and more important, and the need for architectural design method is increasing. In addition, situations in which we have to make the architectural design open are also increasing. For example, in order to make reuse effectively work, we have to share the same architectural assumptions. We also have to share the same architecture to make the system cooperatively communicate in the network environment. All of these recent trends demand systematic and objective methods for architectural design.

There are two types of approach in software engineering field. One is the study on the basic ideas or specific techniques for the field. For example, studies on specific languages, tools, or metrics fall into this category. The other is what we call 'best practice' types of study. In this types of research, we do not mainly focus on the specific techniques or ideas, but we examine how to systematically utilize many techniques we have in hand.

In order to many techniques and ideas in the field work effectively, we need the framework that connects them. Without such a framework, it is difficult to utilize basic techniques and ideas in the actual software development activities. The conceptual framework of software architectural design we have examined in our research is intended to play this role. Using the framework, we can overlook what activities are required in architectural design, how they relate each other, and to where each basic technique is applied.

On the other hand, we also propose some basic techniques that are used in architectural design. As there exists a wide-spectrum of software architectural design, we focus on a specific category of software architectural design, i.e. an architectural design in the early phase of development, in which we examine the direction of system development, mainly from technical view points. These techniques, together with other existing techniques, are systematically combined to support such category of architectural design.

The framework and techniques were examined, in terms of our experiences of the ITS on-board system project. In order to make the framework and techniques effectively work in the real world, we need not only refine them, but also gain experiences of architectural design. In that sense, we hope not only that our research makes contribution to the architectural design, but also that this work becomes the beginning of the next research in the field.

Bibliography

[1] Allen, R.,and Garlan, D.: Formalizing Architectural Connection, Proceedings of the 16th International Conference on Software Engineering, p71-80, May 1994.

[2] Association of Electronic Technology for Automobile Traffic and Driving: Study on ITS On-board system architecture, http://www.jsk.or/jp/eindex.html, 1999.

[3] Barber, K.S and Graser, T.J.:Tool Support for Systematic Class Identification in Object-Oriented Software Architecture, 37th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS-Pacific 2000), 2000.

[4] Bass, L., Clements, P. and Kazman, R.: Software Architecture in Practice, Addison-Wesley, 1998.

[5] Brown, A.W. and Wallnau, K.C.: Engineering of Component-Based Systems,Component-Based Software Engineering - Selected Papars from the Software Engineering Institute, p7-15, IEEE 1996.

[6] Buhr, R.J. and Casselman, R.S.: Use CASE Maps for Object-Oriented Systems, Prentice-Hall, 1996.

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

[8] Chung, L, Gross, D. and Yu, E.: Architectural Design to Meet Stakeholder Requirements, First Working IFIP Conference on Software Architecture (WICSA1), Software Architecture, Kluwer Academic Publishers, p545-564, 1999.

[9] Clements, P.C.: A Survey of Architecture Description Languages, 8th International Workshop on Software Specification and Design, 1996.

[10] Clements, P.C. and Norhrop, L.N.: Software Architecture: An Executive Overview, Component-Based Software Engineering - Selected Papers from the Software Engineering Institute, p55-p68, IEEE 1996.

[11] DeBaud, J. and Schmid, K.: A Systematic Approach to Derive Scope of Software Product-Lines, Proceedings of the International Conference on Software Engineering, 1999.

[12] Gamma, E., et.al.: Design Patterns - Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[13] Garlan, D., Allen, R., and Ockerbloom, J.: Architectural Mismatch: Why Reuse Is So Hard, IEEE Software, Nov. 1995, p17-26.

[14] Garlan, D. and Perry, D.E.: Introduction to the Special Issue on Software Architecture, IEEE Transaction on Software Engineering, Vol.21, No.4. p269-274, April 1995.

[15] Gomaa, H.: Software Design Method for Concurrent and Real-time Systems, Addison-Wesley, 1993.

[16] Gomaa, H, and Manasce, D.A.: Design and Performance Modeling of Component Interconnection Patterns for Distributed Software Architecture, WOSP 2000, 2000.

[17] Hatley, D.J. and Pirbhai, I.A.: Strategies for Real-Time System Specification, Dorset House Publishing, 1988.

[18] Hofmeister, C., Nord, R.L. and Soni, D.: Describing Software Architecture with UML, First Working IFIP Conference on Software Architecture (WICSA1), Software Architecture, Kluwer Academic Publishers, p145-159, 1999.

[19] Jazayeri, M., et.al.: Software Architecture for Product Families, Addison-Wesley, 2000.

[20] Jozwiak, L. and Ong, S.A.: Quality-Driven Decision Making Methodology for System-Level Design, Proc. of EUROMICO-22, 1996.

[21] Kazman, R., et.al.: SAAM: A Method for Analyzing the Properties of Software Architectures, Proceedings of the 16th International Conference on Software Engineering, p81-90, May, 1994.

[22] Kazman, R., et.al.: The Architectural Tradeoff Analysis Method, Proceedings of International Conference on Engineering of Complex Computer Systems (ICECCS), 1998.

[23] Kiczales, G., et.al.: Aspect-Oriented Programming, In proceedings of the European Conference on Object-Oriented Programming (ECOOP), Jun. 1997.

[24] Kishi, T. and Noda, N: Analyzing Hot/Frozen-spots from Performance Aspect, Object-Oriented Product Line Architecture (OPLA), ECOOP, 1999.

[25] Kishi, T. and Noda, N.: Aspect-Oriented Analysis for Product Line Architecture, 1st Software Product Line Conference (SPLC1), 2000.

[26] Kishi, T.: On Software Architecture - Architectural Selection based on AHP -, SIGSE, March, 2001. (In Japanese)

[27] Kishi, T. and Noda, N: Aspect-Oriented Analysis for Architectural Design, International Workshop on Principles of Software Evolution (IWPSE), 2001.

[28] Kishi, T, Noda, N. and Katayama, T.: Architecture Design for Evolution by Analyzing Requirements on Quality Attributes, 8th Asia-Pacific Software Engineering Conference (APSEC 2001), p111-118, 2001.

[29] Kishi, T., Noda, N. and Katayama, T.: Method for Product-Line Scoping based on Decision-Making Framework, 2nd Software Product Line Conference (SPLC2), 2002 (to be appeared).

[30] Klein, M.H., et.al.: Attribute-Based Architecture Styles, First Working IFIP Conference on Software Architecture (WICSA1), Software Architecture, Kluwer Academic Publishers, p225-243, 1999.

[31] Kruchten, P.B.: The 4+1 View Model of Architecture, IEEE Software, Nov. 1995, p42-50.

[32] Kruchten, P.B.: The Software Architect - and the Software Architecture Team -, First Working IFIP Conference on Software Architecture (WICSA1), Software Architecture, Kluwer Academic Publishers, p565-583, 1999.

[33] Krueger, C.W.: Modeling and Simulating a Software Architecture Design Space, CMU-CS-97-158, 1997.

[34] Luckham, D.C., et.al.: Specification and Analysis of System Architecture Using Rapide, IEEE Transaction on Software Engineering, Vol.21, No.4. p336-355, April 1995. http://pavg.starnford.edu/rapide/

[35] Medvidovic, N. and Taylor, R.N.: A Framework for Classifying and Comparing Architecture Desciption Languages, Proceedings of the 6th European Software Engineering Conference, Lecture Notes in Computer Science, 1997

[36] Monroe, R.T., et.al.: Architectural Styles, Design Pattenrs, and Objects, IEEE Software, Jan. 1997, p43-52.

[37] Noda, N. and Kishi, T.: On Aspect-Oriented Design - An Approach to Designing Quality Attributes-, Proceedings of 6th Asia-Pacific Software Engineering Conference, p230-237, 1999.

[38] Object Management Group: OMG Unified Modeling Language Specification (draft) version 1.3 alpha R2, Framingham, MA, 1999.

[39] Perry,D.E. and Wolf, A.L.: Foundation for the Study of Software Architecture, SOFTWARE ENGINEERING NOTES, vol.17, no.4, p40-52, 1992.

[40] Ran, A.: Architectural Structures and Views, Proceedings of the 3rd International workshop on Software architecture (ISAW-3), p117-120, 1998.

[41] Saaty, T.L.: The Analytic Hierarchy Process, McGraw-Hill, 1980.

[42] Selic, B., et.al.: Real-Time Object-Oriented Modelling, Willey, 1994.

[43] Shaw, M., et.al: Abstractions for Software Architecture and Tools to Support Them, IEEE Transaction on Software Engineering, Vol.21, No.4. p314-335, April 1995.

[44] Shaw, M. and Garlan, D.: Software Architecture: Perspectives on Emerging Discipline, Prentice-Hall, 1996.

[45] Soni, D., Nord, R.L., and Hofmeister, C.: Software Architecture in Industrial Applications, Proceedings of the 17th International Conference on Software Engineering, p196-207, April 1995.

[46] Software Engineering Institute,: The Product Line Practice (PLP) Initiative, http://www.sei.cmu.edu/plp/plp_init.html.

[47] Stevens, P.: UML for Describing Product-Line Architecture. Workshop on Object Technology for Product-Line Architecture, ECOOP'99, p109-118, 1999.

[48] Weiss,D.M. and Lai, C.T.R.: Software Product-Line Engineering, A Family-Based Software Development Process, Addison Wesley, 1999.

Publications

[1] Kishi, T. and Noda, N: Analyzing Hot/Frozen-spots from Performance Aspect, Object-Oriented Product Line Architecture (OPLA), ECOOP, 1999. — Make Observation on the impact of infrastructure towards architectural decision.

[2] Kishi, T. and Noda, N.: Aspect-Oriented Analysis for Product Line Architecture, 1st Software Product Line Conference (SPLC1), 2000. —- Propose basic idea of aspect-oriented analysis.

[3] Kishi, T. and Noda, N: Aspect-Oriented Analysis for Architectural Design, International Workshop on Principles of Software Evolution (IWPSE), 2001. —- Discuss architectural design from a point of view of software evolution.

[4] Kishi, T, Noda, N. and Katayama, T.: Architecture Design for Evolution by Analyzing Requirements on Quality Attributes, 8th Asia-Pacific Software Engineering Conference (APSEC 2001), p111-118, 2001. — Propose category of requirements and design method based on the idea.

[5] Kishi, T., Noda, N. and Katayama, T.: Method for Product-Line Scoping based on Decision-Making Framework, 2nd Software Product Line Conference (SPLC2), 2001 (to be appeared). — Propose scoping method based on decision-making framework.

Appendix A

Conceptual Framework of Architectural Design

Here, we show the entire model of our conceptual framework of architectural design. In the following sections, we show class diagrams of UML that depict conceptuarl framework, and give brief definitions and/or explanation for the class and relationship among them.

In the definition, we use the followings;

- Aggregation is referred as $\ll$ *Aggregates* $\gg$.
- Dependency is referred as $\ll$ *Depends* $\gg$.
- Inheritance is referred as $\ll$ *Inherites* $\gg$.
- Link attribute is referred as $\ll$ *LinkAttribute* $\gg$.

A.1 Package Structure

The conceptual framework is consists of three packages (Figure A.1).

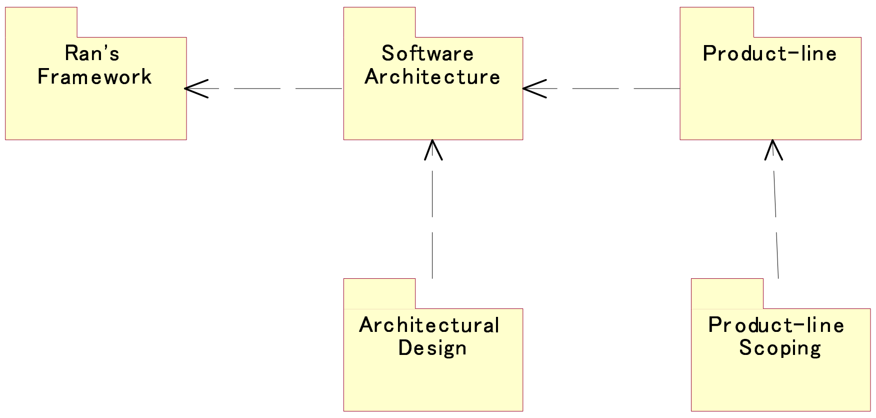


Figure A.1: Package Structure

- **Ran's Framework:** This package includes the model for a revised version of Ran's conceptual framework. As our conceptual framework based on Ran's framework, we define this package to make our conceptual model on it.

- **Software Architecture:** This package includes the model that defines the software architecture and related concepts.

- **Product-line:** This package includes the model that defines the product-line and related concepts.

- **Architectural Design:** This package imports the concepts defined in the package *Software Architecture*, and adds some concepts related to architectural design.

- **Product-line Scoping:** This package imports the concepts defined in the package *Product-line*, and adds some concepts related to product-line scoping.

A.2 Package: Ran's Framework

This package includes the model for a revised version of Ran's conceptual framework (Figure A.2). As our conceptual framework is based on Ran's framework, we define this package to make our conceptual model on it.

Though we have changed some terminologies in order to make consistency with other part of our paper, the basic idea of the model is the same as Ran's original work.

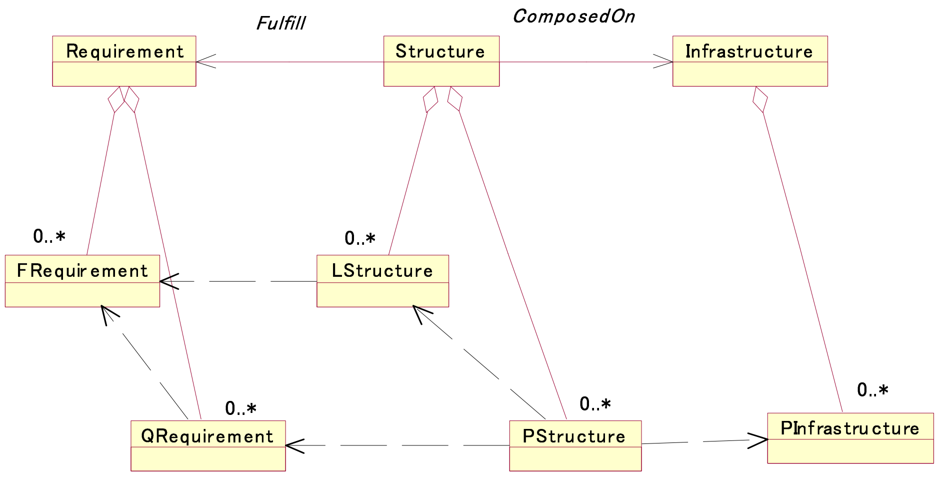


Figure A.2: Package: Ran's Framework

- **Requirement:** *Requirement* is imposed on the software.
  - ≪ *Aggregates* ≫FRequiremnet and (multiple) QRequirements: *Requirement* consists of *Requirements on functionality* and multiple *Requirements on quality attributes*, such as requirements on run-time quality attributes, and requirements on development-time quality attributes.

- **Structure:** *Software structure.* In Ran's framework, this is called as "Architecture". In our model, on the other hand, we use the term "Architecture" in more specific context, as we want to make clear the differences between software architectural design and ordinary software design.
  - *Fulfill* Requirement: *Software structure* is constructed so as to fulfill given *Requirement.*
  - *ComposedOn* Infrastructure: *Software structure* is constructed on the *Infrasturcture*, i.e. *software structure* is the structure of elements provided by *Infrastructure.*
  - $\ll$ *Aggregates* $\gg$LStructure and (multiple) PStructure: *Software structure* consists of *Logical structure* and multiple types of *Physical structure* such as run-time software structure and development-time software structure.
- **Infrastructure:** *Infrastructure* of software.
  - $\ll$ *Aggregates* $\gg$(multiple) PInfrastructures: There are multiple types of *Infrastructure* such as run-time infrastructure and development time infrastructure.
- **FRequirement:** *Requirements on functionality.*
- **QRequirement:** *Requirements on quality attributes.*
  - $\ll$ *Depends* $\gg$FRequirement: Some *Requirements on quality attributes* are defined in terms of *Requirements on functionality*, such as performance of executing a function.
- **LStructure:** *Logical structure* of software. The structure of conceptual entities. This is a special *Software structure* that does not constructed on actual *Infrastructure.* Functionalities of the software is usually defined in terms of *Logical structure.* In Ran's model, this is called as "Conceptual Views".
  - $\ll$ *Depends* $\gg$FRequirement: *Logical structure* is defined based on *Requirements on functionality*, namely it is constructed so as to *Fulfill Requirement on functionality* (inherit from the association between *Requirement* and *Structure*).
- **PStructure:** *Physical structure* of the software, such as run-time structure and development-time structure. In Ran's model, this is called as "Structure".
  - $\ll$ *Depends* $\gg$QRequirement: As each *Physical structure* has an impact on the corresponding quality attributes, *Physical structure* is constructed so as to *Fulfill* the corresponding *Requirement on quality attributes.* (inherit from the association between *Requirement* and *Structure*).
  - $\ll$ *Depends* $\gg$PInfrastructure: *Physical structure* is *ComposedOn* the corresponding *Physical infrastructure*, such as run-time infrastructure and development-time infrastructure (inherit from the asscociation between *Structure* and *Infrastructure.*
- **PInfrastructure:** *Physical infrastructure*,

A.3 Package: Software Architecture

This package includes the model that defines the software architecture and related concepts (Figure A.3).

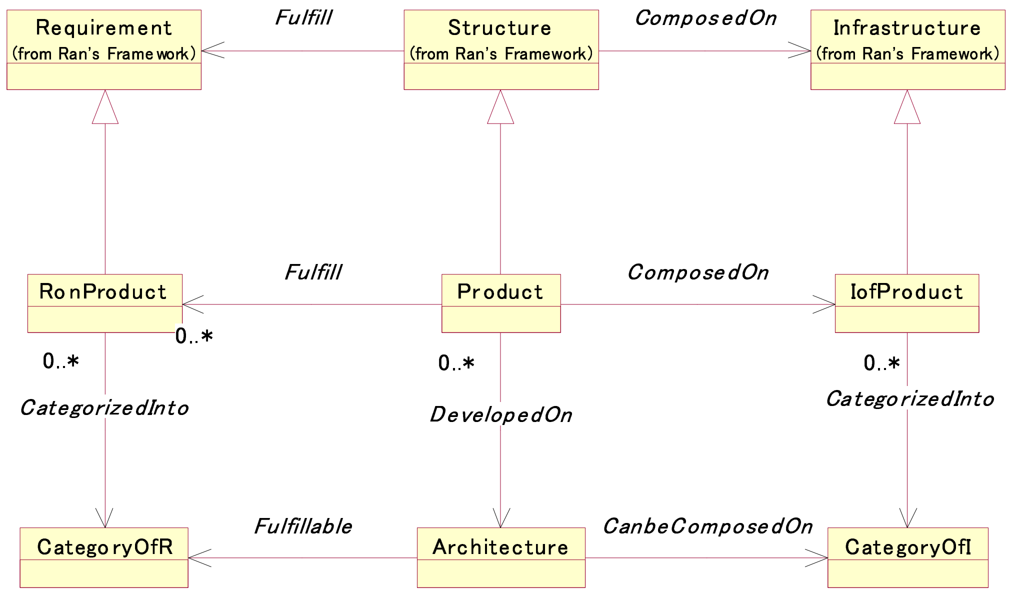


Figure A.3: Package: Software Architecture

- **RonProduct:** *Requirement on software product.*
  - ≪ *Inherites* ≫Requirement : *Requirement on software product* inherits the nature of *Requirement.*
  - *CategorizedInto* CategoryOfR: *Requirement on software product* can be categorized into *Category of requirements.* If the product can be developed on a software architecture, the architecture can be the basis for fulfilling the *Category of requirements.*

- **Product:** *Software product.* Software product has the same characteristics as *Software structure.*
  - ≪ *Inherites* ≫Structure: Software product has the same characteristics as *Software structure.*
  - *Fulfill* RonProduct: This is the inherited association from *Software structure.* We just explicitly show the association.
  - *ComposedOn* IofProduct: This is the inherited association form *Software structure.* We just explicitly show the association.
  - *DevelopedOn* Architecture: *Softawre product* is developed on *Software architecture.* Multiple *Software products* may share the same *Software architecture.*

- **IofProduct:** *Infrastructure of software product.*

  - ≪ *Inherites* ≫Infrastructure: *Infrastructure of software product* has the same nature of *Infrastructure*.
  - *CategorizedInto* CategoryOfI: *Infrastructure of software product* can be categorized into *Category of Infrastructure*.

- **CategoryOfR:** *Category of requirement.* Set of *Requirements on software product* that can be fulfilled by *Software products* developed on the corresponding *Software architecture.*

- **Architecture:** *Software architecture.*

  - *Fulfillable* CategoryOfR: *Category of requirements* is fulfillable by *Software architecture*, i.e. *Requirement on software product* that is categorized into the *Category of requirement*, can be fulfilled by *Software products* developed on the *Software architecture.*
  - *CanbeComposedOn* CategoryOfI: *Software architecture* can be composed on *Category of infrastructure*, i.e. *Software products* developed on the *Software architecture* can be composed on *Infrastructure of product* that is categorized into the *Category of infrastructure.*

- **CategoryOfI:** *Category of infrastructure.* Set of *Infrastructure of product* on which *Software products* can be composed based on the corresponding *Software architecture.*

A.4 Package: Product-line

This package includes the model that defines the product-line and related concepts (Figure A.4).

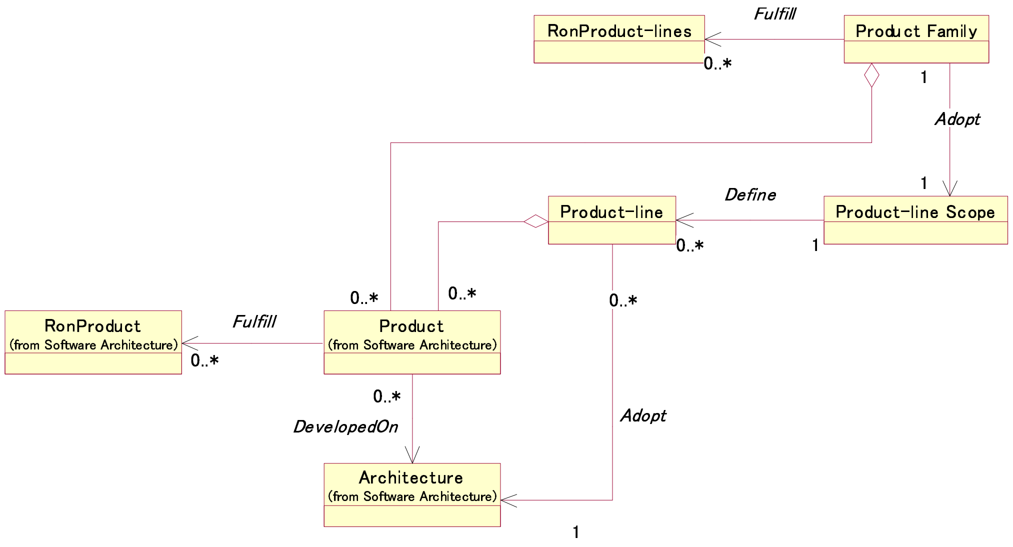


Figure A.4: Package: Product-line

- **Product Family:** *Product family.* Set of products to be developed.
  - ≪ *Aggregates* ≫(multiple) Product: *Product family* consists of multiple *Software products.*
  - *Fulfill* RonProduct-lines: *Product family* is developed so as to fulfill *Requirement on product-lines.*
  - *Adopt* Product-line Scope: *Product family* is developed by adopting a *Product-line scope.*
- **Product-line:** *Product-line* of software products.
  - ≪ *Aggregates* ≫(multiple) Product: *Product-line* consists of multiple *Software products.*
  - *Adopt* Software Architecture: The member of *Product-line* shares a single *Software Architecture.*
- **RonProduct-lines:** *Requirement on product-lines* in terms of software development.
- **Product-line Scope:** *Procuct-line Scope* defines the procuct-lines in a *Product family.*
  - *Define* Product-line. *Product-line scope* determines a set of *Product-line* of software products.

A.5 Package: Architectural Design

This package imports the concepts defined in the package *Software Architecture*, and adds some concepts related to architectural design.

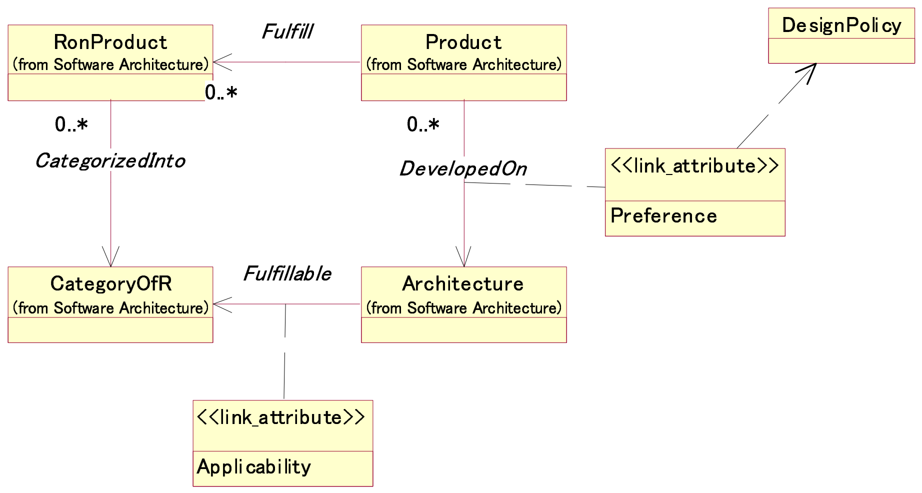


Figure A.5: Package: Architectural Design

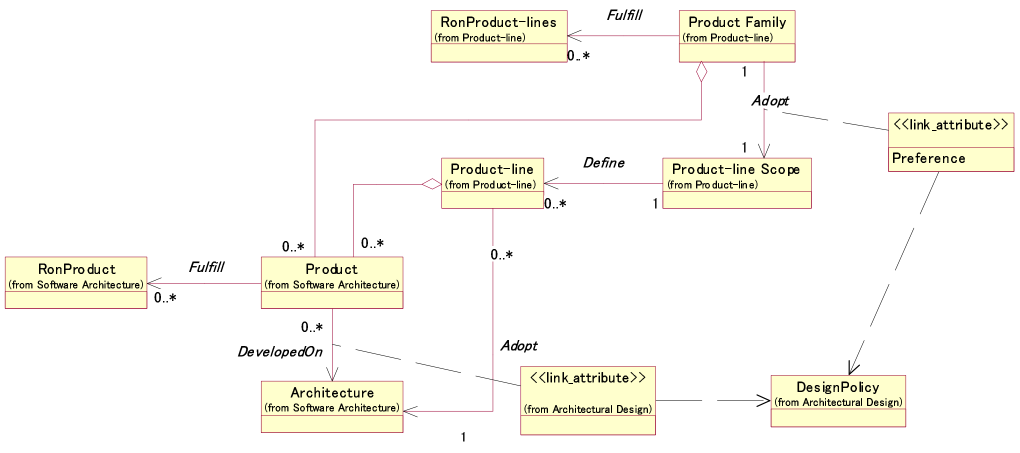
- ≪ *LinkAttribute* ≫**Applicability:** *Applicability of architecture* to *Category of requirements*. This is a link attribute between *Software architecture* and *Category of requirement*.

- ≪ *LinkAttribute* ≫**Preference:** *Preference of architecture* for *Software product* in terms of *Design policy*.

- **Design Policy:** *Design policy* that determines the weight among decision criteria, i.e. quality attributes.

A.6 Package: Product-line Scoping

This package imports the concepts defined in the package *Product-line*, and add some concepts relate to product-line scoping.



RonProduct-lines (from Product-line)
Fulfill
Product Family (from Product-line)
0..*
1
Adopt
<<link_attribute>>
Preference
1
Product-line (from Product-line)
Define
Product-line Scope (from Product-line)
0..*
1
0..*
0..*
0..*
RonProduct (from Software Architecture)
Fulfill
Product (from Software Architecture)
0..*
0..*
DevelopedOn
Adopt
Architecture (from Software Architecture)
<<link_attribute>> (from Architectural Design)
DesignPolicy (from Architectural Design)
1

Figure A.6: Package: Product-line Scoping

- ≪ *LinkAttribute* ≫**Preference:** *Preference of product-line scope* for *Product family* in terms of *Design policy*.

Appendix B

An Example of AHP Calculation

The followings are the steps to determine the preference of the candidates using AHP [41]. In the example in chapter 5, we have determined the preference of the candidates for each product. In this appendix, we will show how to calculate the preference for P1. Other preferences are determined using the same steps.

1. Decide the relative importance among the criteria. We have focused on performance and size. We make pair-wise comparisons between criteria, using the following numbers (Table B.1).

   Table B.1: Numbers Used for Pair-wise Comparison

5	Strongly more important
3	Weakly more important
1	Equally important
1/3	Weakly less important
1/5	Strongly less important

   Table B.2 shows the result of pair-comparisons. The first row shows the relative importance of performance and the second row shows that of size. As P1 is the low-end model and performance is less important than size, we give the following numbers.

   Table B.2: Comparison among Criteria

	Performance	Size
Performance	1	1/3
Size	3	1

   We can obtain the priority vector from above matrix, by calculating the principal eigenvector and normalize them. The following is the result (Table B.3).

2. For each criterion, decide the preference of the candidates. For each criterion, we compare three architectural candidates, and calculate the relative preference (Table B.4, Table B.5).

Table B.3: Preference among Criteria

Performance	0.25
Size	0.75

Table B.4: Comparison among Candidates in Terms of Performance

	A0	A1	A2	Preference
A0	1	1/3	1/5	0.105
A1	3	1	1/3	0.258
A2	5	3	1	0.637

Table B.5: Comparison among Candidates in Terms of Size

	A0	A1	A2	Preference
A0	1	1	5	0.455
A1	1	1	5	0.455
A2	1/5	1/5	1	0.091

3. Obtain overall priorities. Based on the preference among criteria, and preference among candidates, we can calculate overall priorities.

$$0.25 \times \begin{bmatrix} 0.105 \\ 0.258 \\ 0.637 \end{bmatrix} + 0.75 \times \begin{bmatrix} 0.455 \\ 0.455 \\ 0.228 \end{bmatrix} = \begin{bmatrix} 0.367 \\ 0.406 \\ 0.228 \end{bmatrix} \tag{B.1}$$

This gives the preference among candidates for P1 (Table B.6).

Table B.6: Preference among Candidates

	P1
A0	0.367
A1	0.406
A2	0.228