Carnegie Mellon
Software Engineering Institute

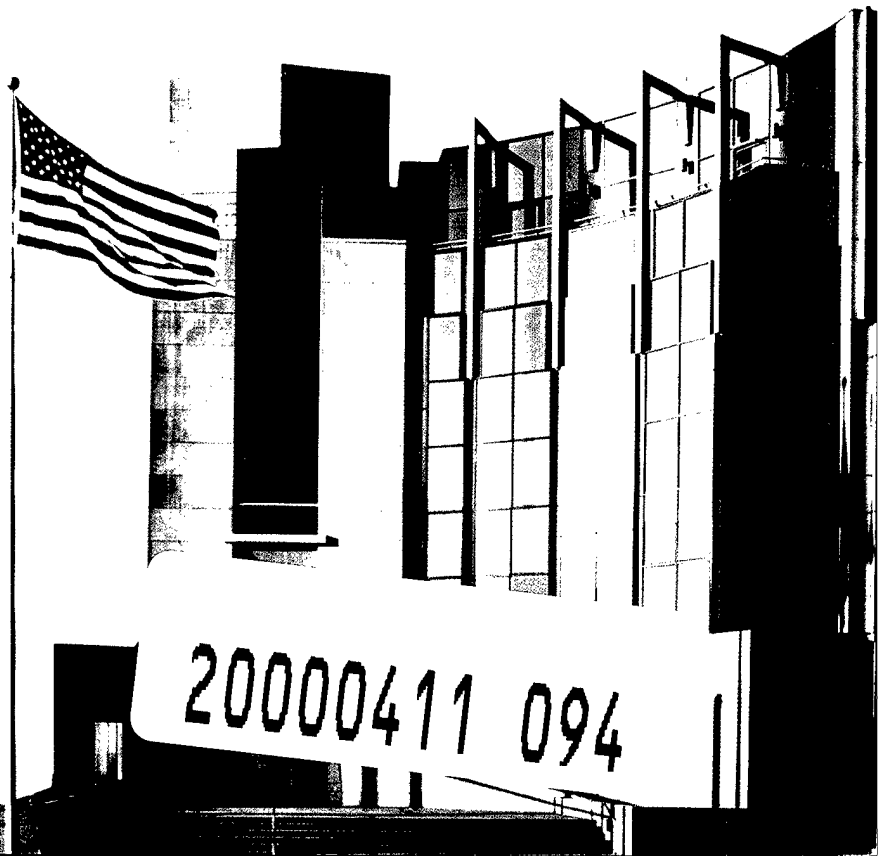
The Architecture Based Design Method

Felix Bachmann
Len Bass
Gary Chastek
Patrick Donohoe
Fabio Peruzzi*

January 2000

* Fellix Bachmann and Fabio Peruzzi are employees of Robert Bosch, GmBH.

TECHNICAL REPORT
CMU/SEI-2000-TR-001
ESC-TR-2000-001



DISTRIBUTION STATEMENT A
Approved for Public Release
Distribution Unlimited

Carnegie Mellon University does not discriminate and Carnegie Mellon University is required not to discriminate in admission, employment, or administration of its programs or activities on the basis of race, color, national origin, sex or handicap in violation of Title VI of the Civil Rights Act of 1964, Title IX of the Educational Amendments of 1972 and Section 504 of the Rehabilitation Act of 1973 or other federal, state, or local laws or executive orders.

In addition, Carnegie Mellon University does not discriminate in admission, employment or administration of its programs on the basis of religion, creed, ancestry, belief, age, veteran status, sexual orientation or in violation of federal, state, or local laws or executive orders. However, in the judgment of the Carnegie Mellon Human Relations Commission, the Department of Defense policy of "Don't ask, don't tell, don't pursue" excludes openly gay, lesbian and bisexual students from receiving ROTC scholarships or serving in the military. Nevertheless, all ROTC classes at Carnegie Mellon University are available to all students.

Inquiries concerning application of these statements should be directed to the Provost, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268-6684 or the Vice President for Enrollment, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268-2056.

Obtain general information about Carnegie Mellon University by calling (412) 268-2000.

Carnegie Mellon
Software Engineering Institute
Pittsburgh, PA 15213-3890

The Architecture Based Design Method

CMU/SEI-2000-TR-001
ESC-TR-2000-001

Felix Bachmann
Len Bass
Gary Chastek
Pat Donohoe
Fabio Peruzzi

January 2000

Program Affiliation

Len Bass, Gary Chastek and Pat Donohoe work in the Product Line Systems program of the Software Engineering Institute. Felix Bachmann and Fabio Peruzzi are employees of Robert Bosch, GmBH.

Unlimited distribution subject to the copyright.

DTIC QUALITY INSPECTED 1

This report was prepared for the

SEI Joint Program Office
HQ ESC/DIB
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

Norton L. Compton, Lt Col., USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

Copyright 2000 by Carnegie Mellon University.

NO WARRANTY

THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Internal use. Permission to reproduce this document and to prepare derivative works from this document for internal use is granted, provided the copyright and "No Warranty" statements are included with all reproductions and derivative works.

External use. Requests for permission to reproduce this document or prepare derivative works of this document for external and commercial use should be addressed to the SEI Licensing Agent.

This work was created in the performance of Federal Government Contract Number F19628-95-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 52.227-7013.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (http://www.sei.cmu.edu/publications/pubweb.html).

Table of Contents

1	**Introduction**	**1**
2	**Quick Overview of the ABD Method**	**3**
3	**Terminology**	**5**
	3.1 Design Elements	5
	3.2 Perspectives and Views	6
	3.3 Use Cases and Quality Scenarios	7
4	**Architecture Design Considerations**	**9**
	4.1 Variabilities and Commonalities	9
	4.2 Software Templates and System Infrastructure	10
	4.3 Architectural Drivers	11
	4.4 Requirements, Quality Attributes, Functionality and Architectural Styles	12
5	**The ABD Method Within the Life Cycle**	**15**
	5.1 Abstract Functional Requirements	16
	5.2 Use Cases	16
	5.3 Abstract Quality and Business Requirements	17
	5.4 Architecture Options	17
	5.5 Quality Scenarios	18
	5.6 Constraints	18
6	**The ABD Method**	**19**
	6.1 Design Elements Defined by the Method	19
	6.2 Order of Generation of Design Elements	20
	6.3 Activities within a Design Element	22
	6.3.1 Divide Functionality	23
	6.3.2 Choose Architectural Style	25
	6.3.3 Allocate Functionality to Style	26
	6.3.4 Refine Templates	26
	6.3.5 Verify Functionality	27
	6.3.6 Generate Concurrency View	28

6.3.7 Generate Deployment View 28
6.3.8 Verify Quality Scenarios 29
6.3.9 Verify Constraints 29
6.4 Next Steps 29

7 Conclusions and Further Work 31

8 References 33

Appendix A - Rose Model of an Example 35
A.1 Rose Constructs 35
A.2 Directory Structure 36

Appendix B. Example 41
B.1 Logical View 41
B.2 Concurrency View 42
B.3 Deployment View 43

List of Figures

Figure 1: Labeling of Decomposition 6
Figure 2: The ABD Method within the Life Cycle 15
Figure 3: Decomposition of System into Design Elements 20
Figure 4: Design Element A Decomposed into Design Elements B and C 21
Figure 5: Steps within Decomposition of a Design Element 22
Figure 6: Defining Logical View 23
Figure 7: Top Level Package Structure 36
Figure 8: Requirement Sub-Structure 38
Figure 9: Subdirectory Structure for the Architecture Directory 40
Figure 10: Example Subsytem Structure 41
Figure 11: Thread View During Initialization 42
Figure 12: Mapping of Design Elements to Units of Deployment 44
Figure 13: Alternative Node Structures 45
Figure 14: Distribution of the Units of Distribution and Processes for the Product Based on Node Structure 1 45
Figure 15: Distribution of the Units of Distribution and Processes for the Product Based on Node Structure 2 46
Figure 16: Communication Mechanisms for the Product Based Node Structure 1 47
Figure 17: Communication Mechanisms for the Product Based on Node Structure 2 47
Figure 18: Virtual Threads vs. Physical Threads 49
Figure 19: Use Case Mapping 50

Abstract

This paper presents the Architecture Based Design (ABD) method for designing the high-level software architecture for a product line or long-lived system. Designing an architecture for a product line or long-lived system is difficult because detailed requirements are not known in advance. The ABD method fulfills functional, quality, and business requirements at a level of abstraction that allows for the necessary variation when producing specific products. Its application relies on an understanding of the architectural mechanisms used to achieve this fulfillment.

The method provides a series of steps for designing the conceptual software architecture. The conceptual software architecture provides organization of function, identification of synchronization points for independent threads of control, and allocation of function to processors. The method ends when commitments to classes, processes and operating system threads begin to be made. In addition, one output of the method is a collection of software templates that constrain the implementation of components of different types. The software templates include a description of how components interact with shared services and also include "citizenship" responsibilities for components.

1 Introduction

Designing a software architecture for a product line of systems is a difficult undertaking. Product lines must be long lived and flexible. Moreover, they must support a set of requirements that are known only in broad scope—the details are unknowable until the actual products are created. Furthermore, the initial stages of architecture design are where the most fundamental design decisions are made; these are the decisions that are the most difficult to correct when they are in error. In order to effectively design a product line architecture, the architect needs a disciplined design method that focuses the creative process; provides a strategy for coping with the uncertainty in requirements; provides guidance in organizing the decisions made during the design process; and makes clear why the steps of the method exist and how they relate to each other.

In this report, we present the Architecture Based Design (ABD) method. The ABD method provides structure in producing the conceptual architecture of a system. The conceptual architecture is one of four different architectures identified by Hofmeister, Nord, and Soni [Hofmeister 00]. It describes the system(s) being designed in terms of the major design elements and the relationships among them. The conceptual architecture represents the first design choices made during a development process. Consequently, it is pivotal to achieving the quality and business goals for the system(s) and in providing a basis for the achievement of the desired functionality.

The ABD method depends on determining the *architectural drivers* for a system. The architectural drivers are the combination of business, quality and functional requirements that "shape" the architecture. With the ABD method, design activities can begin as soon as the architectural drivers have been determined with confidence. This means that the requirements elicitation and analysis activities do not have to be completed (or even very far along) prior to beginning the design. The beginning of design activities does not mean that requirement elicitation and analysis can be discontinued, only that they can go on in parallel with design activities. Especially in situations where determining all of the requirements in advance is not possible, such as for product lines or long-lived systems, the ability to quickly begin design (and hence some level of construction) is important.

The ABD method has three foundations. First is the decomposition of function. For this the method uses well-established techniques based on coupling and cohesion. The second foundation is the realization of quality and business requirements through the choice of architecture style. The third foundation is the use of *software templates*. Software templates have

been utilized in the construction of some systems [Bass 98, Chastek 96]. However, their use is a new concept for design methods, and so we briefly introduce it now.

A software template defines what it means to be a software element of a particular type. This includes patterns that describe how all elements of this type must interact with shared services and the infrastructure. Also, a software template includes "citizenship" responsibilities that pertain to all elements of that type. Examples of such responsibilities are "each element shall log interesting events" and "each element shall provide test points for external diagnostic during run time." Software templates are particularly important in product line systems, since the introduction of new elements is one common technique for managing the specializing of a product line architecture to a particular product.

The ABD method is recursive and the steps within each iteration are clearly defined. Therefore it is always clear during use which design steps have been achieved and which remain. This helps to make the process of architecture design less mysterious.

The ABD method has been used, *in toto,* in a project involving a next-generation automobile and involving personnel from both the Software Engineering Institute (SEI) and Robert Bosch GmbH. It has been used, in part, in other design projects in which the SEI has assisted. Thus, while the ABD method is still evolving, it is sufficiently mature to have been used in actual projects and it has been developed based on experiences in designing large product line architectures.

We begin this report by providing a quick sketch of the ABD method. We continue by dealing with several terminological issues and introducing some basic concepts. We then position the ABD method within the life cycle and describe the method, itself. We also include several appendices. One describes how a design generated using ABD has been represented using an existing commercial design tool; the other gives an example of the interaction of several views generated by the ABD method.

2 Quick Overview of the ABD Method

In this section, we present a quick overview of the ABD method. Everything we discuss in this section will be elaborated in subsequent sections, but we want the reader to have an overview of the method before we plunge into the details.

The input to the ABD method is a list of requirements. These requirements are functional (both abstract and concrete), quality and business (both abstract and concrete), and constraints. A requirement is an item for which the designer has freedom to choose a solution, a constraint is an item that specifies the decision a designer must take. The abstract requirements are used to generate the design; the concrete requirements are used to validate the decisions made as a result of the abstract requirements.

The ABD method proceeds by recursively decomposing the system(s) to be designed. The first decomposition is of the system(s); subsequent decompositions are refinements of the prior decomposition. At each decomposition step, the functional requirements are met by assigning responsibilities to the divisions of the element being decomposed. The quality and business requirements for that element are met by choosing an architecture style, based on the *driving* quality and business requirements, that describes how the divisions relate to one another.

The decomposition is examined from the perspective of three views: logical, concurrency, and deployment. Each view will lead to additional responsibilities for the element being decomposed and these responsibilities must be captured in the decomposition.

The decomposition is also examined from the point of view of the software templates – both to add new items to the templates and to determine where the items in the template will be implemented.

The concrete requirements (functional, quality and business) are used to verify the decisions made during the decomposition. Also, the constraints are examined to verify that none of them has been violated by decisions made during the decomposition.

Design methods such as the ABD method are not intended to replace expert designers. Instead, they are intended to support these designers by providing a structure within which the design can proceed. The ABD method provides a simple and powerful structure. It is simple because the total method can be described as a recursive procedure with a few steps within

the recursion. It is powerful because it provides a method of design that can meet all requirements and validate that those requirements have been met.

3 Terminology

We begin by discussing terms and how we are using them. We then discuss the types of design elements that we are using, how we represent information about these elements as views, and various types of scenarios.

3.1 Design Elements

The ABD method is intended to organize the earliest design decisions. It does not involve commitments to actual software components and classes, nor to organization of the components into processes and operating system threads. We use the term *concrete component* to refer to a component for which a commitment has been made to classes, processes and operating system threads. On the other hand, even though commitments to concrete components are not being made during the method, decisions are being made about division of function and about mechanisms to achieve various quality attributes. The ABD method is a recursive refinement method. The architecture for the system is refined through the method until commitments can be made to software components and classes.

Figure 1 shows the terminology we apply while using the ABD method. At the top level, the *system* is decomposed into *conceptual subsystems* and one or more software templates. At the next level, the conceptual subsystems are decomposed into *conceptual components* and one or more additional software templates.

Because the method is recursive, the steps we apply to the system are the same as the steps we apply to the conceptual subsystems, and are the same as the steps we apply to the conceptual components. We use the term *design element* to refer generically to the system, a conceptual subsystem or a conceptual component. A design element will implement a collection of responsibilities that includes those responsibilities involved in managing concurrency and distribution. A design element has a *conceptual interface* that encapsulates knowledge of data input and output. It is possible that the functions of a particular design element at one stage of the method may be distributed at a later stage in the method into other design elements. The ABD method concludes once decisions begin to be made about classes, methods, processes, and operating system threads. It may be that conceptual components are refined into additional conceptual components. However, our experience has been that once the conceptual components are defined, it is almost always possible to make commitments to concrete components.

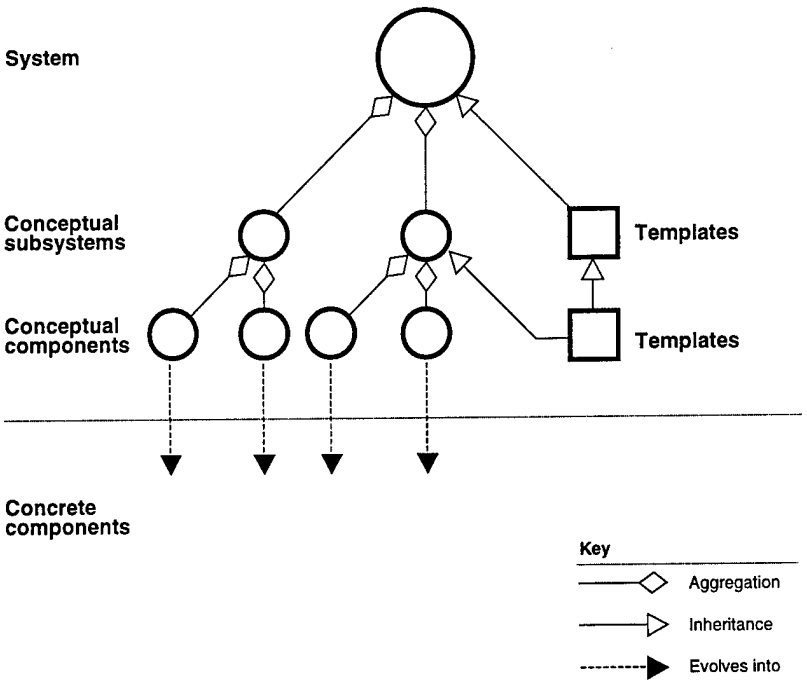


Figure 1: Labeling of Decomposition

3.2 Perspectives and Views

When reasoning about an architecture, it's important to examine it from a variety of different perspectives. This allows the architect to think about different properties of the architecture. For example, a static perspective that displays the organization of the functionality enables certain types of reasoning about development qualities. On the other hand, a dynamic perspective that displays activities that may take place concurrently enables certain types of reasoning about performance.

In the ABD method we view design elements from several different perspectives. That is, a subsystem is not inherently a static architectural element; it is an architectural element that can be viewed from both static and dynamic (or combined) perspectives.

The particular perspectives or views that we choose are similar to those proposed by Kruchten [Kruchten 95]. His views are the logical, process, implementation and deployment views. We use the logical view to record the responsibilities and conceptual interfaces for the design elements. The responsibilities of the design element define its role within the system. They include both functional and quality-oriented items such as "this design element must execute within 50 milliseconds."

We call our second view the *concurrency* view. We use this view to examine the system in light of multiple users, resource contention, start-up and other parallel activities. We use the term *concurrency* in preference to the term *process* to emphasize that no commitment has

been made to processes or operating system threads. The concurrency view will evolve into the process view once these commitments have been made.

The final view that we use is the deployment view. It represents nodes in a computer network; that is, the physical structure of the system. The deployment view is only used for systems that execute on multiple processors. The deployment view displays processors and the networks used to communicate among the processors.

Appendix B provides a sample of how our three views interact.

3.3 Use Cases and Quality Scenarios

Use cases have become an important technique in reasoning about the behavior of a system in a concrete setting. The term has been used in various ways but we use it according to the definition of its authors:

A use case is a piece of functionality in the system that gives a user a result of value. Use cases capture functional requirements [Jacobson 99].

Just as use cases make functional requirements concrete, it is necessary to make quality requirements concrete. It is not meaningful to have a requirement "the system should be easy to modify" since all systems are easy to modify with respect to some sets of changes and difficult to modify with respect to other sets of changes. A concrete form of this requirement is that "it should be easy to add new features of the following type ..." Performance requirements often are stated in terms of latency but without reference to input patterns, and so on. In addition to using use cases to capture the functional requirements we use the technique of defining specific scenarios that embody quality requirements. We call those scenarios *quality scenarios*. Thus, in a typical development, we might have quality scenarios that capture change, performance, reliability and interoperability. We call these change scenarios, performance scenarios, reliability scenarios and interoperability scenarios. The quality scenarios should include both expected and unexpected stimuli. For example, if one expected performance scenario is to estimate the impact of a 10% increase in the number of users per year, one unexpected performance scenario is to estimate the impact of a 100% increase in the number of users per year. The unexpected scenarios may not be realistic but they are useful in determining the boundary conditions within the design.

4 Architecture Design Considerations

In this section, we discuss three considerations that pertain to architecture design and describe how they are related to the ABD method. In particular, we discuss the variabilities and commonalities, software templates and architecture drivers.

4.1 Variabilities and Commonalities

When designing an architecture for a product line of software, it is important to attempt to capture explicitly the envisioned variations that will occur between instances of the product line. The more explicitly these variations can be captured at an early stage, the less turmoil and problems with design occur during product development. These variations can be either coarse or fine grained. For example, a particular automobile may have a radio or a navigation system or both. The product line must be able to support all three configurations. This is an example of coarse-grained variation. On the other hand, the radio may have a dial to control tuning or it may have a digital keypad for entry of a station frequency. This is an example of fine-grained variation. The ABD method is concerned with variation at a granularity that has impact on the conceptual architecture. This is primarily coarse-grained variation but some aspects of the conceptual architecture may exist to allow for fine-grained variation.

Commonalities refers to the fixed points within the variation inherent in a product line. These may be features that are common to all instances of the product line, e.g. if every automobile must have a radio then the radio is "in common." It may also refer to architecture design elements, e.g. if a particular operating system is going to be used for all of the instances of a product line then it is "in common."

Variability can either occur in function (such as the radio examples above) or in platform or environment. An example of platform variability might be the change of an operating system. The ABD method assumes that both kinds of coarse-grained variation (function and platform) are captured during the requirements phase. The mechanism for capturing and representing the commonalities and variabilities is a portion of the requirements process and outside of the scope of the ABD method. Variabilities in platform should be categorized as quality requirements during the requirements phase, and the ABD method has an explicit method to insure the satisfaction of quality requirements. Once the variations are captured, the achievement of this variability becomes the responsibility of the design method. Variability in function is achieved through structural choices made during one step of the ABD method and we will describe this in a subsequent section.

4.2 Software Templates and System Infrastructure

In the introduction, we gave a brief description of the meaning of software template. To repeat, a software template of a particular type enumerates responsibilities for design elements of that type and these responsibilities are one of three kinds:

1. Patterns that describe how this type of design element interacts with services provided to multiple design elements. These are sometimes called *crosscutting services*. For example, if there is a diagnosis service, then all conceptual components that provide data to the diagnosis service should use the same protocol.

2. Patterns that describe how this type of design element interacts with the infrastructure. For example, it may be that every conceptual component must have an install and deinstall method that will be invoked by the infrastructure. Placing this information in the component template records this knowledge.

3. Citizenship responsibilities. Those responsibilities that every design element of a particular type must provide are placed in the software template. For example, the requirement to handle errors in a consistent fashion may be a responsibility of particular types of design element. Placing this information in the template records it and simplifies the identification of common methods for meeting these responsibilities.

Software templates serve several purposes: They are a repository for certain components that can be reused within the system, they are an aid to integration, and they provide the basis for constructing a skeletal system. The nature of software templates enables their role as repositories; they identify patterns of behavior that transcend particular components. Consequently, the implementation of the responsibilities that make up the template is likely to be reusable within the system.

Software templates are an aid to integration because of the categorization of design elements into types. The concrete realization of these design elements into components will consequently also be typed. Consider, as an example, a conceptual component A and a collection of conceptual components of type B. Assume A interacts with multiple instances of components of type B. The pattern of interaction between A and each of the instances of type B will be the same. Thus, integrating A with one instance of type B will simplify the integration of A with other instances of type B. Some integration problems are caused by data inconsistencies and others are caused by the pattern of interaction. Those problems caused by the pattern of interaction need only be solved once but can be applied to multiple instances.

A template captures the interactions of design element types with both the infrastructure and the shared services. Thus, in aggregate, an implementation of the templates together with the shared services and the infrastructure represent a skeleton of the system(s) being designed. By a *skeleton*, we mean an implementation of the necessary infrastructure of the system or systems with no or very little application functionality.

A skeletal implementation such as this allows subsequent incremental addition and roll out of functionality. It also embodies the support necessary for the functional variations within a product line. The development pattern, from the customer's perspective, is that there is a period of design where nothing is visible, followed by the development of the skeleton where extremely limited functionality is visible, followed by features implemented in an order that supports the business goals of the system.

Other implementation orders are also possible. The definition of software templates and what it means to be a design element of a particular type does not directly support these other implementation strategies, but neither does this definition hinder other implementation orders.

Some quality attribute modeling techniques such as those used in performance analysis and availability analysis depend on knowing the patterns of interactions of the various architectural element types without reference to specific functionality. The definition of software templates determines the patterns of interaction and allows these models to be developed.

4.3 Architectural Drivers

Architectural drivers are the combination of functional, quality and business requirements that "shape" the architecture. If the driving requirements can be met, then the system can be satisfactorily designed. At the top level, the drivers can be determined by looking at the purpose of the system and critical business needs. At more detailed levels, the drivers are determined by the architect from the requirements on the particular conceptual subsystem or conceptual component. Some examples of system level drivers based on the purpose of the system are

- The purpose of a flight simulator is to train aircrews and this dictates both high fidelity of the simulation and real-time performance.
- The purpose of an air traffic control system is to perform real-time control of enroute aircraft and this dictates high availability and reasonably stringent performance.
- The purpose of some financial systems is to transfer money or orders from one place to another and this dictates high security and high availability.

Another source of drivers might be the business goals and background of the organization constructing the systems. For example

- The organization wishes to develop a product line, and this dictates a concern for generality that might not occur in a single product architecture.
- The organization has an investment in prior systems in the domain. This dictates reusing components from prior systems and, consequently, either reusing the architecture from prior systems or developing an architecture that accomodates the legacy components.
- The organization wishes to develop a particular competence, such as in Web-based database access. Consequently, the architecture for the next system will be strongly influenced by that desire.

- The organization has established a business relationship with a particular software supplier. Consequently, the architecture will reflect the use of the components furnished by that supplier.
- The particular product being developed will have time to market, size of market and lifetime of the product constraints. Consequently, the amount of generality or specificity built into the architecture will be affected.
- The product must interoperate with other products. Consequently, the character of the interfaces being exposed will be affected.
- The organization has employees with particular talents who must be utilized. Consequently, the architecture will be designed with identifiable components suited to the talents of these employees.

Observe that the drivers do not depend on the details of the functional requirements but on an abstraction of the functional requirements. That is, in the flight simulator case, whether the aircraft being simulated has two engines or four is not an architectural driver—but the achievement of real-time performance in the face of large amounts of data movement is. This means, as we observed before, that design activities could begin as soon as the architectural drivers have been determined with confidence. This could be very early in the life cycle depending on the familiarity of the architecture team with the domain.

Determination of the architectural drivers is not always a top-down process. It may involve detailed investigation of particular aspects of the requirements in order to understand the architectural implications. For example, navigation systems depend on use of CDs for storage of map data and there are two different standards for storage of the maps. These two standards use different approaches that have been optimized for different query types. A requirement might dictate, for example, that performance differences in the two different formats be masked from the end user. This would lead to a detailed investigation of the formats, in order to determine the best architectural approach to achieve the requirement. Whether this particular requirement is a driving requirement depends on the solution. If the solution is far reaching then it is a driving requirement; if the solution is easy to achieve and does not affect the overall architecture, then it is not.

4.4 Requirements, Quality Attributes, Functionality and Architectural Styles

An *architectural style* consists of a collection of component[1] types together with a description of the pattern of interaction among them [Bass 98]. Example component types are client, server, layer, and process. As such, the component types have functionality to the extent necessary to implement the patterns of interaction, but they have no application functionality associated with them. One of the important steps in the ABD method is to choose a dominant architectural style for a particular set of requirements. In the previous section we discussed

[1] This is a different use of the word *component* than is used when referring to conceptual or concrete components. The word *component*, when used by the architecture community, is a primitive that describes any connection of coherent computation.

the concept of architectural driver. In this section we discuss how architectural drivers enable the choice of architectural style through the interplay of requirements, functionality and quality attributes.

Consider, again, the flight simulator example we introduced above. In this example, the architectural driver was the requirement to process large amounts of data with hard real-time performance constraints. A driving requirement may involve any combination of function, quality and business requirements. In the flight simulator example, the functional facet of this requirement is the movement of large amounts of data; the quality facet of this requirement is hard real-time performance. One architectural style that results from this driving requirement is a real-time scheduling strategy. This strategy depends on both the functional and the quality facets of the driving requirement. That is, if there were not large amounts of data to use then real-time performance could be achieved under other scheduling disciplines. If real-time performance were not an issue, then the large amount of data movement could be achieved under other scheduling disciplines. It is the combination of the two that drives the choice of style.

Furthermore, once the style has been chosen, it doesn't implement any application functionality. The functionality must be divided and allocated to instances of the component types. Thus, in the flight simulator example, what are being scheduled are specific computations. The number and function of these computations must be determined independently from the non-driving requirements. The criteria for determining the number and function of these computations will differ from the driving requirements (although they are constrained by them). As in the flight simulator example, there may be computations for each engine and these computations must exist in component types defined by the real-time scheduling style. The fact that each engine has a separate computation is determined by requirements for modifiability. We use the term *division of functionality* to refer to the division of the functional requirements in response to specific criteria.

In the ABD method, two closely related and intertwined steps are the division of functionality and the choosing of an architectural style. The architecture results from allocating the functional divisions to the instances of component types defined by the style. It is this architecture that can be analyzed for how well it achieves various properties.

5 The ABD Method Within the Life Cycle

Figure 2 shows the placement of the ABD method within the life cycle. We assume a requirements phase to be at least partially complete, although we do not prescribe a particular method of requirements gathering, organization or tracking. We do require certain output from the requirements phase, including functional requirements, quality and business requirements, and constraints. The output of the ABD method is a collection of conceptual components in three views, including: the assumptions that can be made about each conceptual component; a collection of software templates; and those concrete implementation decisions that have already been made. We maintain the concrete implementation decisions as additional constraints.

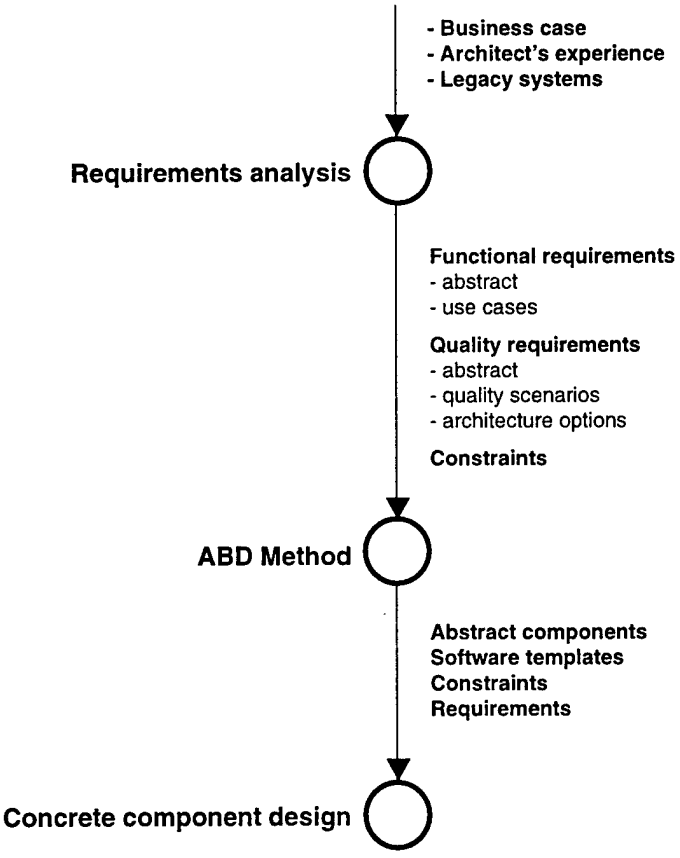


Figure 2: The ABD Method within the Life Cycle

All decisions made during the execution of the ABD method must be recorded as well as the rationale for those decisions. This allows traceability for a decision should it become necessary to revisit it. Appendix A of this report documents the use we made of Rational Rose as a tool for recording these decisions.

The input to the ABD method consists of

- abstract functional requirements, including the identification of variabilities and commonalities
- use cases (concrete functional requirements)
- abstract quality and business requirements
- quality scenarios (concrete quality and business requirements)
- architecture options
- constraints

We now describe the assumed outputs of the requirements phase (the inputs to the ABD method).

5.1 Abstract Functional Requirements

The ABD method assumes that one of the outputs of the requirements phase is an abstract characterization of the functional requirements, together with a characterization of the coarse variability within those requirements. It is important to consider all of the end users when capturing the requirements.

A variety of end users are typically associated with a particular system. Different system administrators (database, system, network) may be end users. Maintenance technicians may be end users for systems such as computers embedded in automobiles or aircraft. An end user is anyone who uses a system while it is in execution.

Associated with the abstract functional requirements is a characterization of the commonalities and the coarse variabilities associated with these requirements. Understanding the dependencies among the requirements is also important for the design.

The functional requirements must be captured at an abstract level because detailed requirements for the individual products in a product line may not be known until the products are constructed. The abstract capture of the requirements provides categories for the detailed requirements when they become known.

5.2 Use Cases

As we defined above, a use case is a concrete expression of an interaction between one (or more) end users and the system. For the purposes of use cases, another system that must in-

teroperate with the system being designed can be considered an end user. Use cases are easy to generate and it's possible to have hundreds or even thousands of them. Use cases require analysis, however, and so their number should be limited. Only the most important ones are useful during architecture design; the use cases that are generated should be grouped and prioritized in order to identify the most important. The remainder can be generated at any portion of the design process.

5.3 Abstract Quality and Business Requirements

The quality and business requirements for the system(s) to be constructed should be enumerated. Each quality requirement should include a specific stimulus as well as the response that is desired. Quality requirements such as "The system shall be modifiable" are essentially meaningless, since every system is modifiable for some set of modifications and difficult to modify for others. A requirement for modifiability should characterize the set of modifications such as "The system (a flight simulator) shall be modifiable with respect to changes in the aircraft being simulated."

The distinction between business requirements and quality requirements is not clear. Many important business goals for the architecture will translate into abstract quality requirements. For example, the goal that the architecture supports a product line will translate into a collection of extendibility requirements. We lump together quality and business requirements in the ABD method, since the source of a requirement is not important to the method.

5.4 Architecture Options

For each quality and business requirement, possible architecture options that could enable the satisfaction of the requirement should be enumerated. For example, if the requirement were to support a variety of different user interfaces, then one architecture option would be to separate the user interface into a separate component. Another requirement might be to remain independent of a particular operating system. Then an architecture option would be to have a virtual operating system layer that receives all operating system invocations and translates them to the current operating system. No decisions among the options are made at this point; the intent is to enumerate all possible options. This list of architecture options is where the architect's experience base is applied. The options are mechanisms or patterns for solving particular quality problems. These mechanisms or patterns would come, ideally, from a handbook; absent such a handbook, they come from the architect's background.

This enumeration of options, logically belongs to the ABD portion of the process and not to the requirements phase. However, since it is natural to begin to enumerate possible solutions while generating quality and business requirements, we list this as an output of the requirements phase. This enumeration will be extended as additional options are discovered during the design phase. Architectural options are a subset of what Hofmeister, Nord, and Soni call "strategies."

5.5 Quality Scenarios

Just as use cases make concrete functional requirements, quality scenarios make concrete quality requirements. Quality scenarios are very specific expansions of the quality requirements. While a quality requirement from the above example might be "modifications to a flight simulator that reflect modifications to the underlying aircraft should be easy," a quality scenario might be "modify the system to change the rotor in an engine."

As with use cases, quality scenarios are easy to generate and it is possible to generate many of them. They should be prioritized so that during the design phase, only the most important will need to be examined.

5.6 Constraints

A constraint is a design decision that is pre-specified. The design process consists of making decisions. Some of these decisions can be derived directly from the business goals without consideration regarding its impact on design. For example, if an organization has a large investment in a particular middleware product, that product may be chosen without reference to other decisions. During requirements capture, constraints come primarily from the business goals surrounding the system. One special case of these constraints is determined by legacy systems, which we will now explore in detail.

Few systems today are designed without reference to existing systems. Frequently the new system must interoperate with existing systems. In other cases the new system is replacing the existing system, and must reuse as much as possible from the existing system. In any event, these legacy systems are external to the system being currently designed and some of their characteristics must be considered during the design process.

To the extent that legacy systems will affect the current design, it is important to understand their structure and the techniques they employed to solve problems. Business reasons may require use of components from legacy systems or from particular commercial vendors. Such requirements will tend to become constraints. Business reasons may also require that competitive systems be examined for relevant features or structural aspects.

While utilizing the ABD method, the architect may make decisions that do not immediately appear in the design. For example, the architect may decide that a particular real-time operating system must be used. This is not a business decision but a technical one. Although this is technically a design decision and not a constraint, we use the list of constraints as a place to record this type of decision. From the point of view of the design method, it does not matter whether a constraint results from a business reason or a technical decision.

6 The ABD Method

Now we turn to the description of the method itself. We begin by describing a view of the artifacts that emerge from the method. We then turn to a more process oriented view.

6.1 Design Elements Defined by the Method

The ABD method is based on decomposing the overall system. Figure 3 repeats the elements of this decomposition. Every system consists of an application portion and an infrastructure portion. Although the boundaries between these portions is not always clear, a design must consider both the application and the infrastructure on which it executes. The ABD method captures these two fundamental portions by viewing the system as the combination of an application portion and the infrastructure. Both of these portions contribute to the decomposition and their subsequent definition. The top-level decomposition of the system is into conceptual subsystems. Associated with these conceptual subsystems are subsystem templates. The conceptual subsystems together with the subsystem templates aggregate into the system. Each conceptual subsystem has its own responsibilities and is considered according to the three views: logical, concurrency, and deployment.

The conceptual subsystems are, in turn, decomposed into conceptual components. As with conceptual subsystems, each conceptual component is considered from three views. Associated with the conceptual components are component templates.

In the figure, we show concrete components as being the next decomposition after the conceptual components. A concrete component reflects a commitment to a software element such as a class. Depending on the size of the system being designed, it is possible to have additional design elements between the conceptual components and concrete components but, for the size of the systems on which the ABD method has been used, this additional level of design elements has not been needed.

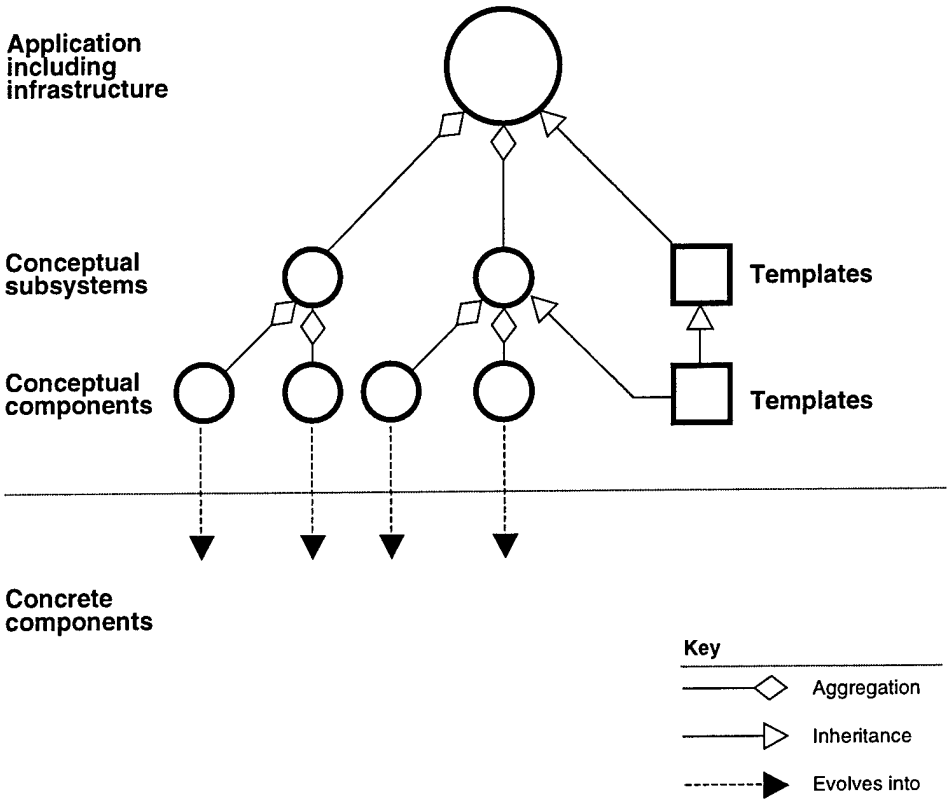


Figure 3: Decomposition of System into Design Elements

6.2 Order of Generation of Design Elements

Figure 3 shows the design elements and their relationships after exercising the ABD method. It does not show how this tree may be traversed. There is a wide variety of schemes for traversing the tree.

One possibility is to traverse the tree breadth first— define all of the conceptual subsystems prior to decomposing any of the conceptual subsystems and then define all of the conceptual components prior to making commitments to concrete components.

Another possibility is to traverse the tree in some depth for one or more of the conceptual subsystems and fill in the other conceptual subsystems in a later stage in the process.

It is always the case that more detailed understanding of a particular aspect of the design may cause reconsideration of prior decisions. In our case, this means that during the generation of the conceptual subsystems, insight into the requirements might be gained that causes new requirements to be added or existing requirements to be modified. During the generation of the conceptual components, decisions made during the definition of the conceptual subsystems may be revisited in the light of new information. This type of re-investigation in light of

better understanding is a characteristic of all designing efforts including those using the ABD method.

Also, a detailed investigation of some aspect of the design may be done at any point in the process—in order to understand the design options and the implications of choosing a particular option. This is appropriate. The only consideration is that decisions made must be recorded in the proper place.

Some of the considerations in determining the path to traverse the tree for a particular development are

- *the knowledge of the domain.* If the architect has extensive knowledge of the domain then exploration will not be as necessary.
- *the incorporation of new technology.* If new technology is to be used as middleware or for the operating system, then prototypes will need to be constructed. This will both aid understanding of the capabilities and limitations of the new technology, and give the architecture team experience in using the technology.
- *the personnel on the architecture team.* People with specific expertise may be engaged to explore a particular portion of the tree in some depth.

Now consider Figure 4. This figure shows a design element A that is decomposed into two smaller design elements B and C. We will use this figure to discuss the interplay between requirements and responsibilities. We first refer solely to A. There are certain requirements that A must satisfy (functional, business and quality). These requirements are satisfied by the responsibilities of A (functional and those quality attributes that are dependent only on A).

During the decomposition of A into B and C, then, the responsibilities of A are decomposed into requirements on B and C. During the reasoning about B, responsibilities are assigned that enable the satisfaction of its requirements.

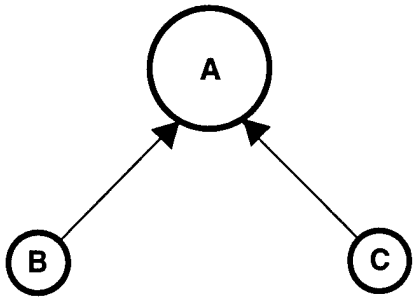


Figure 4: Design Element A Decomposed into Design Elements B and C

6.3 Activities within a Design Element

Up to this point, we have discussed paths for traversing the tree of design elements in order to decompose them, but we have yet to discuss how to decompose a design element. According to the method, we begin the decomposition of each design element with a set of requirements (both functional and quality), a template that pertains to that design element, and a set of constraints. The output of the process for the design element is a list of children design elements, each having a set of requirements, templates that pertain to it, and a set of constraints. Figure 5 shows the usual sequence of the steps of the decomposition of a design element with a feedback loop between the verification and the definition of the logical view (as we have shown), but it is possible to begin with the other views. The feedback loops will help ensure that all of the views are considered. The architect may wish to begin with the deployment view, for example, if the system(s) being designed has unusual connectivities. Although not shown here, it is also possible to have feedback back to the logical view during the definition of any of the other views. The location of the feedback loops is a function of the type of system(s) being designed.

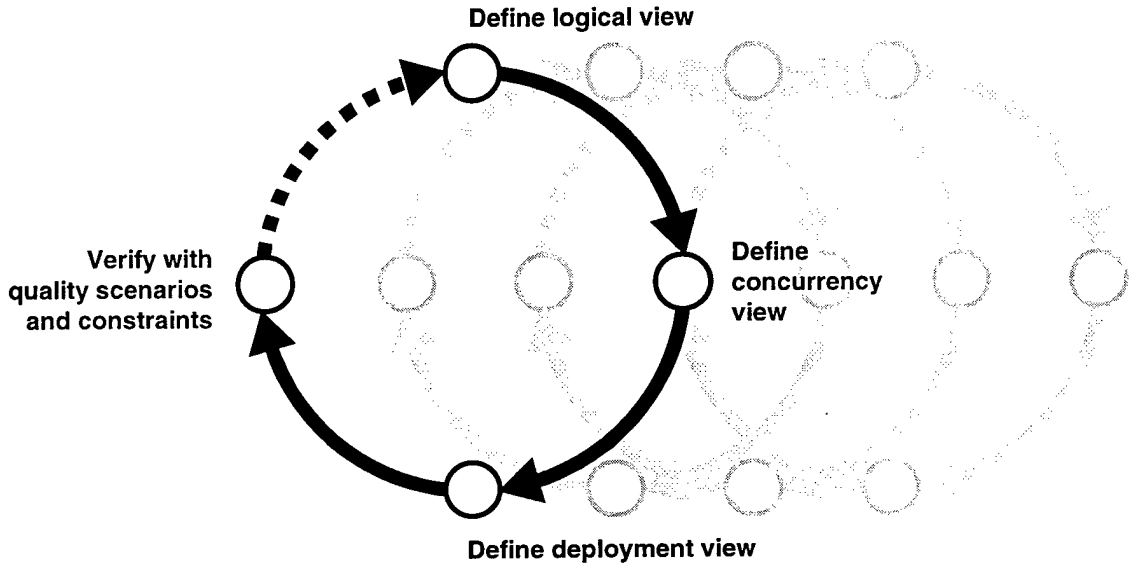


Figure 5: Steps within Decomposition of a Design Element

The steps are as follows: define the logical view, define the concurrency view, define the deployment view, and verify. At each step, there are likely to be additional functions that must be managed by the logical view, requiring that the decisions made thus far be revisited.

Figure 6 shows the steps involved in the definition of the logical view. In this case, function is decomposed, a base architectural style is chosen, and the function is allocated to the style.

These three steps are intricately intertwined, as we have already discussed. Then the software templates are refined and the result is verified by exercising use cases and change scenarios.

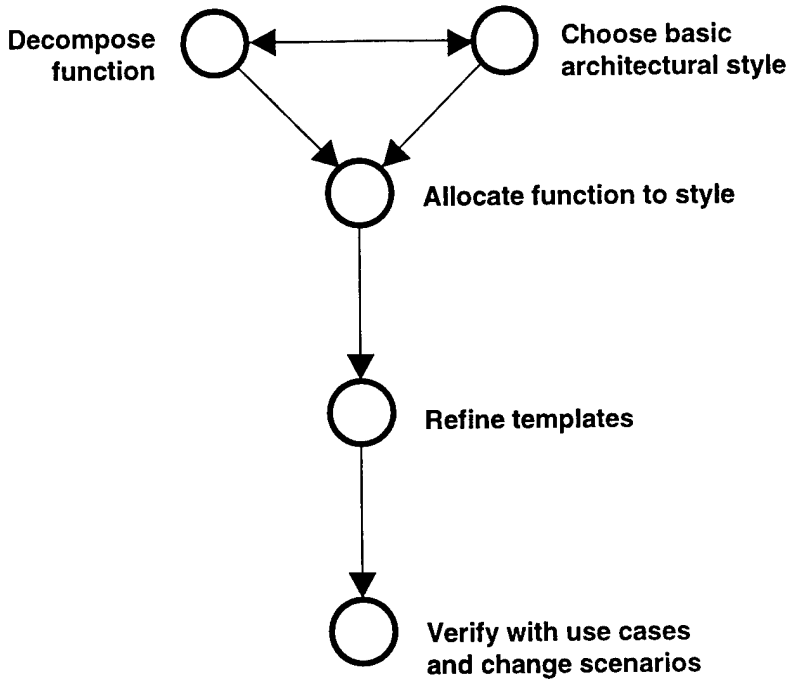


Figure 6: Defining Logical View

As with all of the steps in the ABD method, executing one step may produce insight that will cause reconsideration of prior steps. For example, when the concurrency view is generated, additional functionality may be identified that was not considered in the requirements. This functionality may cause a reconsideration of the basic architectural style for the design element; it may have to be allocated and verified with use cases. Furthermore, discussions that occur during one step may uncover information that pertains to a future step. For example, during discussion of functional decomposition, items may be discovered for the templates. These items should be recorded as they are discovered.

We now describe these steps in more detail.

6.3.1 Divide Functionality

A design element has a set of responsibilities. These responsibilities should be divided into groups. The goal of the division is to have each group represent distinct elements within the architecture. The divisions may need to be refined further, even to the extent of breaking up

design elements. The criteria for this division will depend on the qualities that are important to a particular design element. There may be multiple divisions that are based on different qualities. Below we provide examples of such differing qualities.

If the quality influential in the division is modifiability, as is usual in product lines, then the groups of responsibilities are chosen based on several criteria:

1. Functional coherence. Requirements grouped together should exhibit low coupling and high cohesion. This is a standard technique for decomposing function. The use cases can be used to investigate the cohesion and coupling. The quality scenarios dealing with change can also be used to investigate cohesion and coupling.

2. Similar patterns of data or computation behavior. Those responsibilities that exhibit similar patterns of data and computation behavior should be grouped together. What it means to exhibit similar behavior depends on the particular domain for which the system(s) are to be used. If data acquisition is a responsibility then a similar pattern might refer to the sampling period. If a particular responsibility is computationally expensive then it should be grouped with other computationally expensive responsibilities. Responsibilities that access a database in a similar fashion should be grouped together.

3. Similar levels of abstraction. Responsibilities that are close to the hardware should not be grouped with those that are more abstract. On the other hand, responsibilities that exist at the same level of abstraction should be grouped together.

4. Locality of responsibility. Those responsibilities that provide services to other services should not be grouped with purely local responsibilities.

Other criteria for modifiability exist in addition to those stated above. Also, there will be responsibilities that do not fit neatly into any grouping. If these are substantial enough they should be assigned to their own group. If not, they should be assigned to another group, even when they do not create a good match.

If the quality influential in the decomposition is performance, then the division is based on minimizing the amount of data movement among the pieces (assuming a fixed set of algorithms) and frequency of data calculations (assuming periodic processes).

If the investigation into the decomposition causes a detailed investigation of a particular aspect, then decisions made during the detailed investigation should be recorded. They will either be recorded as a responsibility of one of the design element or they will be recorded as additional constraints that must be satisfied.

The goal of this decomposition is to produce groups of functionality of sufficient granularity so as to 1) represent a decomposition of the design element responsibilities while 2) keeping

the number of groups intellectually manageable. Nominally, between 3 and 15 groups should be produced.

Associated with each group of responsibilities is a characterization of the coarse-grained variability and the dependencies of that group. Also associated with each group of responsibilities is an enumeration of the information requirements of that group and the information produced by that group. This data flow view of the decomposition leads to the conceptual interface in terms of its information needs and production. The information flow and the design elements with which the current design element interacts can be determined by the generation of a diagram. A tool that supports the design process should generate this automatically. The diagram would show the entities external to the design element (both other design elements and actors external to the system) with which the design element interacts.

Finally, it is important to recall that not all of the requirements are known and, hence, the list of responsibilities for any particular design element is incomplete. Thus, the groups of responsibilities should be broad enough to provide a location for additional requirements that might be added to the system either during or after initial development.

6.3.2 Choose Architectural Style

Each design element has a dominant architectural style or pattern. It is the basis for how the design element achieves its responsibilities. The dominant style is not the only style within the design element; it may be modified to achieve particular goals. The choice of architectural style is based on the architectural drivers for this design element. Thus, the process is to determine the architectural drivers for this design element and from the architectural drivers and consideration of the functional decomposition, to determine the dominant architectural style.

It is not always the case that a documented style can be chosen as the dominant style. In this case, the architect should make up a new style that fits the needs at hand. Sometimes, the responsibilities of a design element consist of relatively independent collections and a dominant style can not be discerned. In this case, a style of independent filters with connections from outside the design element may be appropriate.

Once the dominant architectural style has been chosen, it should be adapted based on the quality requirements that pertain to this design element and the architecture options identified to satisfy the quality requirements. That is, examine each quality requirement, and determine whether it is relevant to the design element being decomposed. If it is, choose one of the options associated with the quality requirement and apply it to the style chosen for the design element.

It is desirable to use the same option whenever a particular quality requirement is relevant to multiple design elements and so the option should be associated in the design record with the

design element. This will enable revisiting of the decision regarding what option to use once the design has been further elaborated.

The result of choosing and refining an architectural style is a collection of (architectural) component types. There may be types such as "client" without reference to either the functionality to be computed by the client, or to how many instances of the client may exist. Such determinations will be made in the next step. Some component types, especially those resulting from quality requirements, may have associated functionality, such as "virtual device." This association should be retained for use in the allocation step.

The choice of an architectural style for a design element is a fundamental choice that relies heavily on the architects' experience in design. The style chosen may be a run-time style such as client-server; a development-time style such as layered, or may have aspects of both, such as in a three-tiered architecture. In any case, the choice of style may introduce additional functionality that must be placed into the functional groups.

6.3.3 Allocate Functionality to Style

The choice of architectural style yields a collection of component types. The number and function of each of these types must be determined. This is the purpose of the allocation. The groups of functionality resulting from the decomposition of function should be allocated to the types of components resulting from the determination of style. This involves determining how many instances of each type of component will exist and what the functionality of each instance will be. The components resulting from this allocation will be the candidate child design elements for the design element being decomposed.

The conceptual interface for each design element is also identified. The interface consists of both the information that the design element needs and produces (identified during the functional decomposition) and the data and control flow information needed by each component type within the defined architectural style.

It is in the iteration of these three steps (divide functionality, choose style, and allocate functionality to style) that tradeoffs are made among the various quality attributes. The designer must determine whether the compromises that have been made are adequate. The validation steps that follow the definition of the logical view provide concrete evidence of these compromises.

6.3.4 Refine Templates

The design element being decomposed has a collection of templates that pertain to it. These are inherited from its parents up the hierarchy. At the initiation of the method, the system has no template. Responsibilities are added to the templates as they are refined. These responsibilities must be implemented by a concrete component at some point in the design process.

Thus, one step in exiting the ABD method is to assign responsibilities in the template either to concrete components or to classes that are inherited by concrete components.

For each responsibility in the existing templates, the following questions are asked:

- Are there aspects of this responsibility that are handled by the child design elements or should this responsibility remain in its current location? There are two possible answers:
  1. The responsibility should remain in its current location. In this case, nothing is done.
  2. The responsibility will be divided between those aspects that are satisfied at the current location and those that will be assumed by the child level design elements. The appropriate templates are updated to reflect this division.

The responsibilities of the child design elements are also examined to determine which should be added to the template. These are

- those responsibilities that can be shared across some subset of the child design elements rather than locally within each child, or
- responsibilities that every child must manage in some fashion such as error handling, logging of activities, or providing test points for external diagnostics.

Finally, the responsibilities of the templates are examined to determine whether any additional responsibilities need to be added to any of the design elements anywhere in the system. That is, identify any crosscutting services that exist at this level. The templates contain both what it means for a design element to be a good citizen (e.g., log certain type of information) as well as those responsibilities that should be shared rather than handled locally. Each type of responsibility may require additional support functionality, (e.g., having the actual writing to a log be a responsibility of a particular design element). This support functionality, once it has been identified, must be allocated.

6.3.5 Verify Functionality

The use cases are exercised to verify that they can be achieved through the proposed structure. Additional responsibilities for the child design elements will likely be determined through the exercise of use cases. However, if use cases were used extensively in the functional decomposition, then very few additional responsibilities might be found.

Change scenarios are also exercised at this point since the difficulty of performing a change is based on the division of functionality.

From this type of verification, the design is shown to cover the requirements (through the use cases) and support the modifications (through the change scenarios).

6.3.6 Generate Concurrency View

Thus far we've considered the logical view of the child design elements. The concurrency view also must be considered. The purpose of examining the concurrency view is to determine what activities might be carried on in parallel. These activities must be identified and points of spawning new threads, synchronization and resource contention discovered.

The examination of the concurrency view is in terms of *virtual threads.* A virtual thread is a single path of execution through a program, a dynamic model, or some other representation of control flow [OMG 99]. This should not be confused with the operating system thread that includes additional implications of address space and scheduling strategy. An operating system thread is the conjunction of several virtual threads, but every virtual thread is not necessarily an operating system thread. Virtual threads are used to describe a sequence of activities and so synchronization or resource contention will be between multiple virtual threads.

Use cases which examine the effects of having two users, the effects of parallelism on the activities of one user, or the effects of start-up and shutdown, are useful in thinking about the concurrency view.

Discovery of points of synchronization and resource contention may add new functionality. For example, a resource manager may be needed to manage contention. In such cases, the new functionality must be allocated as a responsibility to a design element.

6.3.7 Generate Deployment View

If multiple processors are used in the system, then issues arise from deploying design elements to separate processors. These issues are examined using the deployment view. We examine the effect of the network on the virtual threads. A virtual thread may travel via the network from one processor to another. We use the term *physical thread* to describe the threads that exist on a particular processor. That is, a virtual thread is composed of the concatenation of physical threads. Through this view, we may discover requirements for synchronizing physical threads on a single processor and for handing off a virtual thread from one processor to another.

Questions should be asked during the generation of the deployment view about the impact of the network on data transmission, latencies, and on the synchronization of activities.

We use the concept of *unit of deployment* within the deployment view. A unit of deployment is the smallest design element that can be allocated to a processor. Exactly what this means depends on the granularity of the design elements. A decision must be made as to which level of design element constitutes a unit of deployment. That is, what is the coarsest design element that will not be allocated or split among different processors? If the design elements are coarser than the smallest unit of deployment, then the design element may be split across two

processors. In this case, it may imply a division of functionality or it may imply multiple instances of the design element.

If the design element is a unit of deployment or a refinement of a unit of deployment then generating the deployment view means that decisions must be made about deployment, although not necessarily about packaging in processes.

6.3.8 Verify Quality Scenarios

Once the three views are available, the quality scenarios should be applied to the children design elements being generated. For each quality scenario, ask whether it is still possible to satisfy the scenario. Each quality scenario includes a quality attribute stimulus and a desired response. Consider the decisions made so far in the design and determine whether it is still possible to achieve the quality scenario.

In the event of a negative answer, then either the decisions that have been made should be reconsidered, or the architect must accept the failure to realize one of the quality scenarios. The rationale for accepting a failure to realize one of the quality scenarios should be recorded.

6.3.9 Verify Constraints

The final step is to verify that none of the constraints have been violated by the decisions made in this step. That is, for each constraint, ask the question "is the achievement of this constraint still possible?"

A negative answer is treated in the same fashion as a negative achievement of a quality scenario. The rationale must be documented and the decisions that led to the constraint being imposed must be re-examined.

6.4 Next Steps

The process we have described is applied to produce a collection of design elements according to the sequence determined as being appropriate for the project under consideration. The ABD method is used to generate the high-level architecture. Once commitments begin to be made to concrete components, to process and thread allocations, and to specific deployments, then decisions such as class and object structure, data types, and interface specifications must be made. The ABD method has no features that pertain to this class of decisions. Another method should be used that addresses these issues. The outputs of the ABD method that should be considered during more detailed design are the design elements, the templates, the quality requirements, the quality scenarios, and the representation of commonalities and variabilities.

The refinement of the commonalities and variabilities should continue only until decisions can be made as to the appropriate technique for managing variation. Replacement, delegation,

overloading, and parameter definition are all techniques for managing variation. We will not examine those techniques here. For a further discussion see Clements [Clements 99].

7 Conclusions and Further Work

In the introduction to this paper we identified certain attributes that we strive to establish in a design method: discipline, capability to deal with uncertainty in requirements, guidance in organizing the decisions made, and clear rationale for activities within the method. The ABD method is our attempt to achieve these goals. The discipline emerges from both the recursive nature of the method and the steps within each iteration. The capability for dealing with uncertainty in requirements is achieved by isolating uncertainty to the details of the requirements and achieving certainty in architectural drivers and abstract functional decomposition. The guidance in organizing the decisions and the clear rationale for activities are related, and we hope that this exposition demonstrates that there is a rationale for each step of the method.

The ABD method grew out of earlier work in the analysis of software architecture. This work is embodied in the Architecture Trade off Analysis Method (ATAM) [Katzman 99]. The goal of an ATAM is first to understand the architecture and secondly to analyze, based on that understanding. The design perspective is to propose a design (or partial design) and then to analyze based on the proposed design. The analysis portion is common to both methods. The reader familiar with analysis techniques will recognize the use of quality scenarios. The question then becomes: is there virtue in performing an ATAM when the architecture was designed using the ABD method? The answer reflects the two purposes of an architecture analysis: to view the design with fresh eyes and evaluate the design using a fixed method. Viewing the design with fresh eyes is always appropriate, and any architecture, no matter how designed, can benefit from being evaluated in this fashion. Since the ABD method includes mini-ATAMs at each iteration (the verification steps), there are no methodological virtues of performing an ATAM on a design generated with the ABD method.

An Attribute Based Architecture Style (ABAS) is an architecture style with attached descriptions of how to reason about an instantiation of that style [Klein 99]. ABASs (as they are developed) should clearly be incorporated into the ABD method. The step of the ABD method that chooses an architecture style and the step that decomposes function should both be influenced by the use of ABASs. The verification steps should be influenced by the reasoning associated with ABASs. We are planning some exercises where the ABD method will be used in conjunction with ABASs that are also being developed.

The ABD method is one example of how understanding of architecture and the influences that drive architecture design can be exploited to help the development process. There are many other portions of the general development process that could be improved by taking advantage of an understanding of architecture. The exciting challenge for the future is to de-

termine those portions of the development process and to find the right methods to exploit the architecture understanding.

8 References

[Bass 98] Bass, L.; Clements, P.; & Kazman, R. *Software Architecture in Practice*. Reading, MA: Addison Wesley, 1998.

[Chastek 96] Chastek, G. & Brownsword, L. *A Case Study in Structural Modeling* (CMU/SEI-96-TR-035, ADA324233). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1996. Available WWW <URL: http://www.sei.cmu.edu/publications/documents/96.reports/96.tr.035.html

[Clements 99] Clements, P. & Northrop, L. "A Framework for Product Line Practice." Version 2.0, July 1999. Available WWW <URL: http://www.sei.cmu.edu/plp/frameworkv2.7.pdf

[Hofmeister 00] Hofmeister, C.; Nord, R.; & Soni, P. *Applied Software Architecture*. Reading MA: Addison Wesley, 2000.

[Jacobson 99] Jacobson, I.; Booch, G.; & Rumbaugh, J. "The Unified Process." *IEEE Software*, (May/June 1999) 96-102.

[Kazman 99] Kazman, R.; Barbacci, M.; Klein, M.; Carriere, S.J.; & Woods, S.J. "Experience with Performing Architecture Tradeoff Analysis." 54-63. *Proceedings of ICSE99*. Los Angeles, CA, May 1999.

[Klein 99] Klein, M.; Kazman, R.; Bass, L; Carriere S.J.; Barbacci, M.; & Lipson, H. "Attribute-Based Architectural Styles, Software Architecture." 225-243. *Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)*, San Antonio, TX, February 1999.

[Kruchten 95] Kruchten, P. "The 4+1 View Model of Architecture." *IEEE Software* 12,6 (1995): 42-50.

[OMG 99] Object Management Group. "OMG Unified Modeling Language Specification" (draft) Version1.3 alpha R2, Framingham, MA: January 1999.

Appendix A - Rose Model of an Example

The previous section describes the process of generating and refining design elements. Recording the decisions made during architecture design is critical because of the sheer number of these decisions. In this section we describe our use of the Rational Rose '98 modeling tool to perform this recording. We describe the directory structure for the recording, the stereotypes we used, how traceability was maintained, and, in general, how Rose can be used to support this design effort. No commercial tool supports the concepts necessary to perform the method we have described but we include this section to show that commercial tools can be used, even if the use is with a certain amount of force fitting. We make no claim that Rose is the best tool for our use or that we made the best use of Rose, but only that such a tool can be used to support the ABD method.

We begin this section by discussing the concepts available in Rational Rose. We next discuss how the concepts of the ABD method are mapped to the available concepts. Then we discuss the directory structure we used.

A.1 Rose Constructs

Rose has five fundamental constructs that appear to be relevant to its use for the ABD method. In fact, only four of these constructs were actually used. The five constructs are components, classes, associations, stereotypes, and packages. Beside these, Rose offers a variety of diagrams that can be used to show various aspects of the system.

The component construction is intended for describing language-specific elements such as header files. Since the ABD method terminates prior to the introduction of language elements, we did not use this construction at all.

We used the class construction to capture the concepts in the ABD method that consist of textual descriptions. These concepts included design elements, interfaces for design elements, both conceptual and concrete requirements, constraints, and architecture options.

We used the association construct to represent different types of relationships between classes. Each association was given a name that was descriptive of its purpose.

We used the stereotype construct to identify types of classes or associations. Thus, a class might have a name such as "operating system" and a stereotype such as "conceptual subsystem."

Finally, the package construct can be decomposed. That is, a package can have sub-packages and so forth. We used the package construct to contain our directory structure.

A.2 Directory Structure

Rose pre-specifies the initial level of the directory structure using the name "View." At this initial level, the system provides the following choices:

- Use Case View
- Logical View
- Component View
- Deployment View

There is no "Process View."

The Component View and the Deployment View are restricted to contain the Rose construct "components." Since we did not use the component construct, we did not use the Component View or the Deployment View.

Our basic directory structure (with the exception of use cases) consisted of packages within the logical view. Figure 7 gives the top level of our structure.

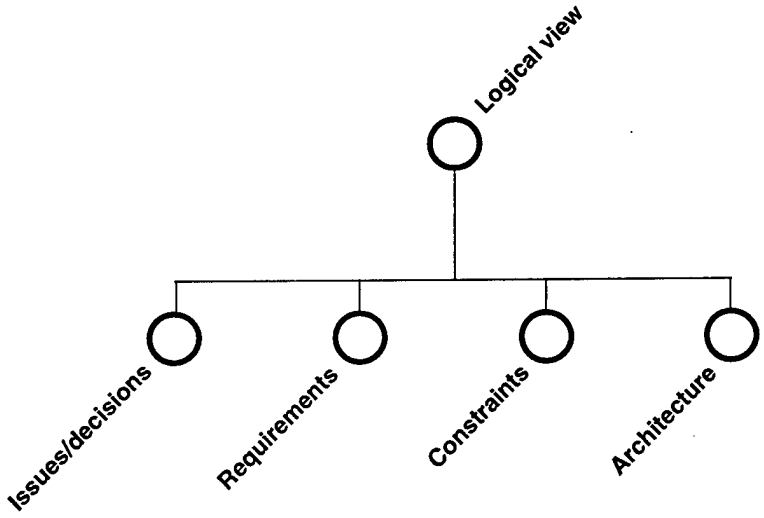


Figure 7: Top Level Package Structure

Of these four sub-packages, the issues/decisions and the constraints packages contain textual lists. That is, within the issue/decisions and the constraint sub-packages are named classes

that represent one item on the list (such as a constraint). Within the class is text that describes that item on the list.

The issues/decision list is not mentioned in our description of the ABD method since it isn't properly a portion of the method. This list was used to maintain issues that arose and also global decisions that were made that were not specific to one or several design elements. Examples include a decision concerning a minimum legal configuration within the product line, and an issue regarding whether a bus of a particular type has the capacity to support a particular protocol. Decisions include a rationale, and both decisions and issues are stereotyped classes (<<Decision>>, <<Issue>>) linked using associations (<<requires>>) to the design elements that either generate the issues or that resolve them. Decisions can have dependency relations (<<DependsOn>>) among each other. To illustrate those associations, diagrams that show important aspects may be included in the issues/decision package.

Items on the constraints list are also linked to the design elements where the constraints are satisfied. Issues also can depend on each other. This is documented by using a dependency relation (<<DependsOn>>). To illustrate important aspects of constraints and their relations, diagrams may be included in the constraints package.

When the number of elements in the lists is large, the issues/decisions and the constraints packages may be further structured using packages.

Figure 8 shows the directory sub-structure under the requirements package.

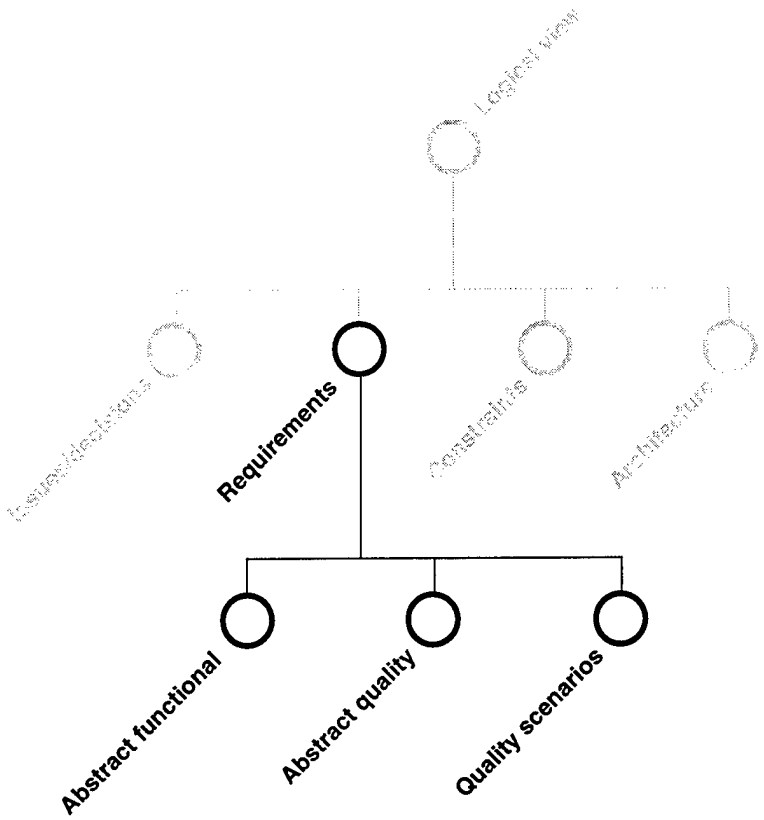


Figure 8: Requirement Sub-Structure

The manner in which the conceptual functional requirements are recorded depends on how they are represented. We assume that the sub-packages *abstract functional* and *abstract quality* have a list of packages, each named with the name of the requirement/quality. Every package includes at least a class with the name of the requirement/quality. The description of this class is the description of the requirement. The package may include diagrams to show more complex information. That information could be an enumeration of architectural components or a pattern that could be used to support a requirement/quality.

The classes in the sub-packages of the *abstract functional* packages that hold the description of the requirement have the stereotype <<Requirement>>. Diagrams included in sub-packages can be enumerations of architectural components or patterns that could be used to support the requirement.

The sub-packages of the Abstract Quality package also contain at least a class (<<Quality>>), named with the name of the quality and a more detailed description of that quality. The sub-

package may contain diagrams illustrating possible architectural options, which are a collection of components and/or pattern of interaction.

The sub-package Quality Scenarios contains a list of use cases (<<Quality Scenario>>), each named with the name of the scenario, and also includes the description of that scenario. Each use case can have a diagram attached to it that shows particular architectural components that are affected by the specific scenario. Those diagrams normally are created later in the development process. At the time when quality scenarios are created, not a lot of information about architectural components is available; information is added to this package as the scenarios are applied to various design elements.

Use cases, which describe functional requirements in a more concrete way, would logically fit into this structure, but we used the Use Case View (at the top level) to maintain use cases. Each use case is in a package, and contains a use case diagram that represents classes of various types linked together with associations. The classes that appear in a use case diagram are either design elements or a representation of external entities such as actors.

Figure 9 shows the first level of the structure of the architecture section of this directory. There is a package for each of the conceptual subsystems, a package for the templates generated during the decomposition into conceptual subsystems, a package for the concurrency view, and a package for the deployment view.

Inside the package for each of the conceptual subsystems is a class (<<Subsystem Representative>>) that enumerates the subsystem's responsibilities and conceptual interface (class diagram that shows the usage) and a package for each of the children of that conceptual subsystem.

Inside the template package is a set of classes that enumerates the responsibilities of the template and a diagram that shows the relations between those classes. Classes that symbolize patterns have the stereotype <<Pattern>>. The template package also includes sub-packages for each child of that template.

Inside the concurrency view package is a collection of concurrency use cases for the conceptual subsystems. The diagrams attached to those use cases show threads and their interactions and synchronization points. Threads are modeled as stereotyped use cases (<<Thread>>) and synchronization points are stereotyped classes (<<Synchronization>>).

Inside the deployment view package, are use case diagrams showing possible scenarios of the deployment view, as well as a collection of deployment use cases. The diagrams show possible deployments. They include nodes (stereotyped classes <<Node>>) and their connections with each other. The assignment of elements described in the conceptual subsystems to the nodes is described by using the realize relation stereotyped with <<deployed>>.

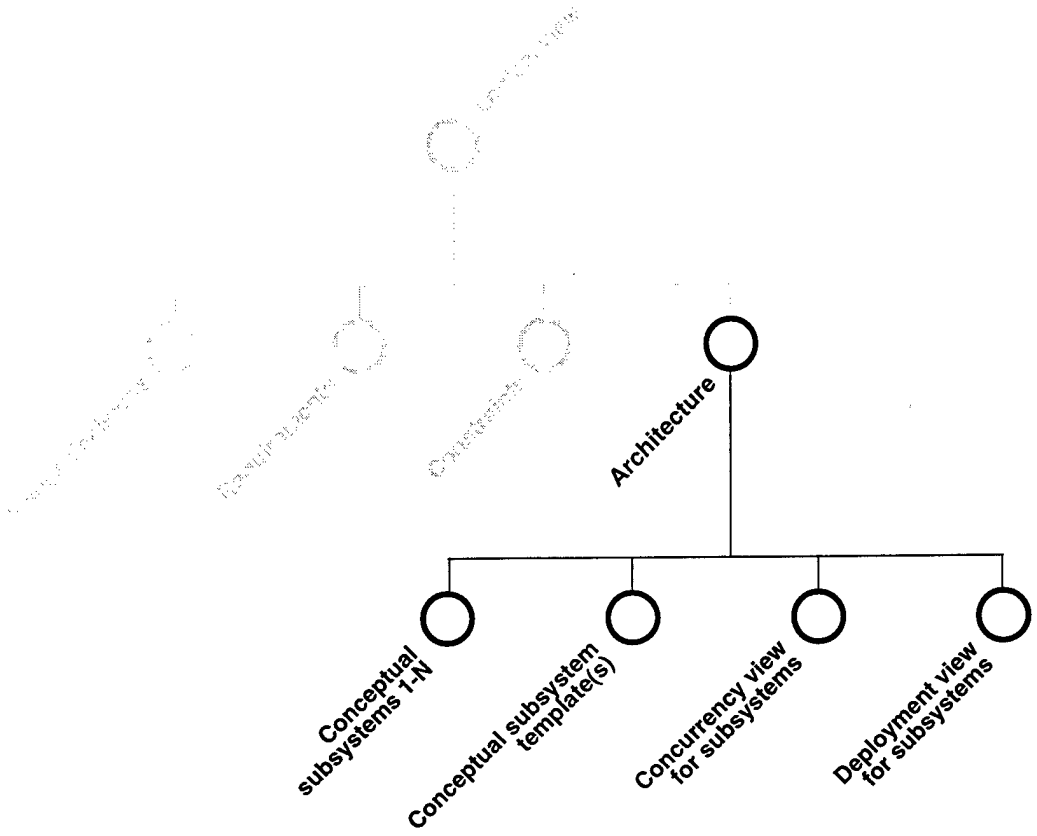


Figure 9: Subdirectory Structure for the Architecture Directory

Appendix B - Example

This appendix provides an example of the interaction of the three views that the ABD method uses: the logical, the concurrency and the deployment.

B.1 Logical View

We now describe the notation that we use for the logical view. A rectangle is used to represent the design elements of conceptual subsystem or conceptual component. We use ovals to refer to specific responsibilities that are contained within the design element. The specific responsibilities are just identified within the design element and do not actually emerge as Rose entities (unless they become design elements in a further decomposition). These responsibilities are very important in reasoning about the design elements, however, and we explicitly identify them for this discussion.

We assume the logical view shown in Figure 10, organized as layers. Within Rose, nested icons such as those presented here are not possible, and so the nesting is accomplished through the use of sub-packages.

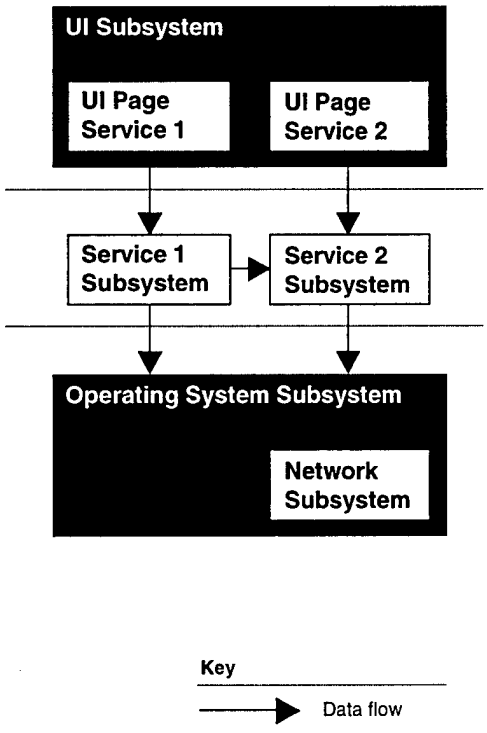


Figure 10: Example Subsytem Structure

In Figure 10, we identify four conceptual subsystems: UI, Service 1, Service 2 and Operating System. Within two of these subsystems (the UI and the Operating System Subsystems), there are smaller components. Each of the conceptual subsystems and conceptual components has an enumerated list of responsibilities. This list, initially, is based on the functional requirements and the functional use cases.

B.2 Concurrency View

Once the initial logical view has been defined, the concurrency view is examined through use cases. Figure 11 shows a use case for initialization that identifies several specific responsibilities that must occur during initialization. The arrows in Figure 11 represent threads.

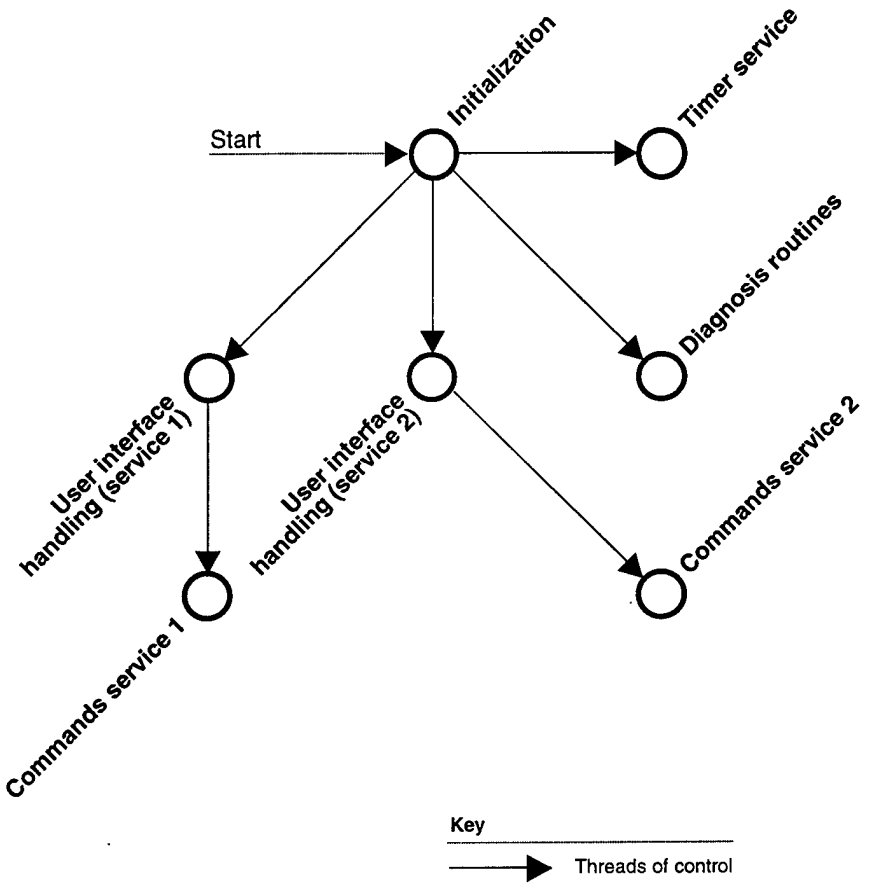


Figure 11: Thread View During Initialization

We assume that products, after the start-up, go through an initialization phase in which some background activities, such as the Timer Service and Diagnosis, are started. We assume, as well, that the user interfaces must be ready to get input from the users. We also assumed that two different—independent and running in parallel—interfaces are active. The last assump-

tion is that the user interfaces must be always ready to get input from the outside. This means the user interfaces must run in parallel with any commands previously executed.

The ovals, as we said, represent specific responsibilities that were identified during the analysis of this use case. Three of these responsibilities (timer, diagnosis, and initialization) were not previously identified (in our hypothetical development of responsibilities) as responsibilities of the any design element. We add these responsibilities to those of the operating system.

Notice that the concurrency view depicts various threads of control through the design elements. That is, the design elements and their responsibilities are the basis of each view. Reasoning about the view will result in adding responsibilities to the design elements.

B.3 Deployment View

We use the notation of a diamond to represent a unit of deployment and the notation of a line with rectangles above it to represent a processor configuration.

Now assume that the following *Units of Deployment* have been identified:

- UI Page Service 1
- UI Page Service 2
- Service 1
- Service 2
- Operating System

Although this is a direct mapping of the logical view, in general, direct mapping will not be the case. It could be that instances of a particular element of the logical view would each be deployed to different processors. It also could be that elements of the logical view would need to be decomposed before units of deployment of are identified.

Figure 12 shows how the various design elements are mapped to the units of deployment.

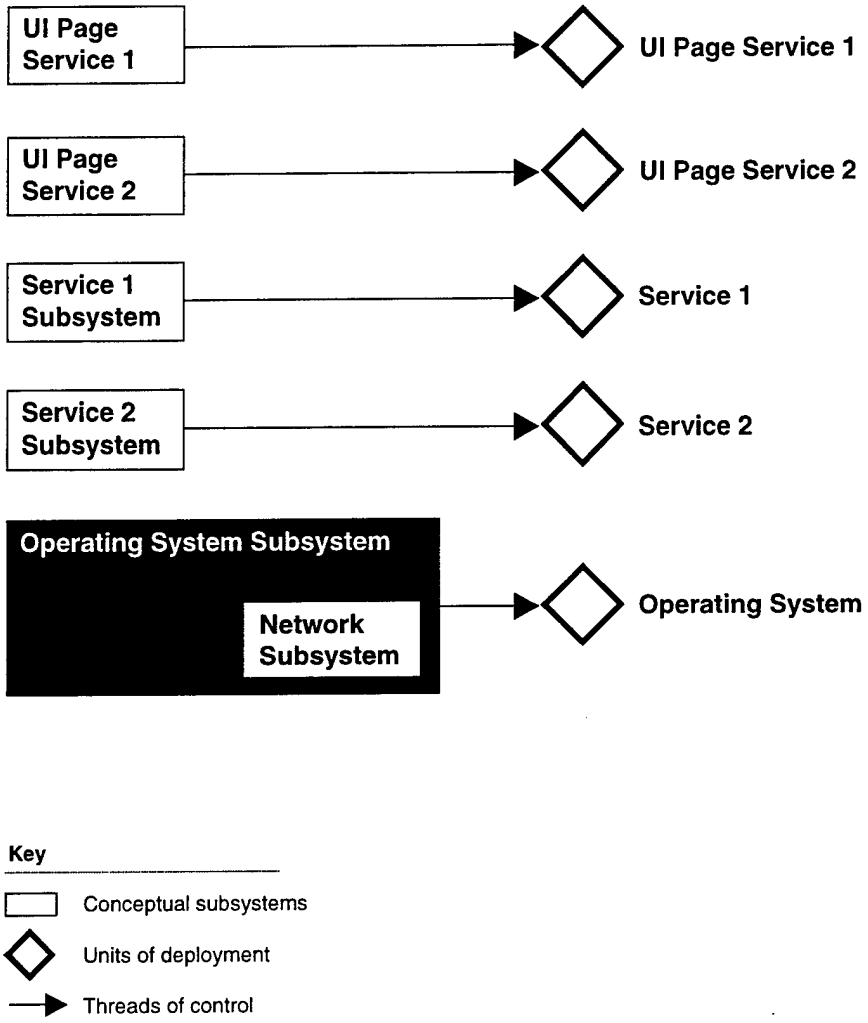


Figure 12: Mapping of Design Elements to Units of Deployment

For the remainder of this example, we will suppose that two products must be generated. The two products provide the same functionalities and should use the same software but they have to run on different hardware platforms. The node structures for the products are shown in Figure 13. The symbology indicates that the two processors in Node Structure 1 are equivalent. In Node Structure 2, there are two equivalent processors and one smaller processor.

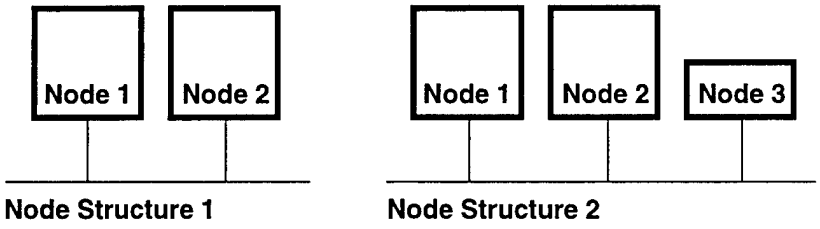
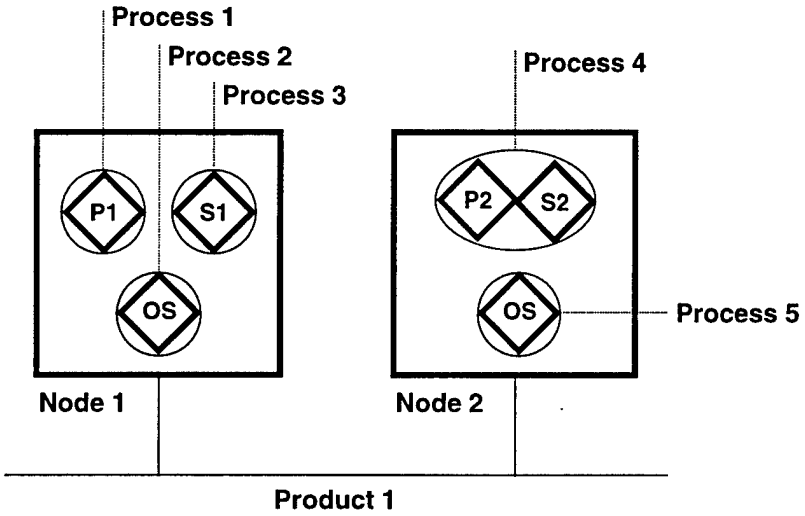


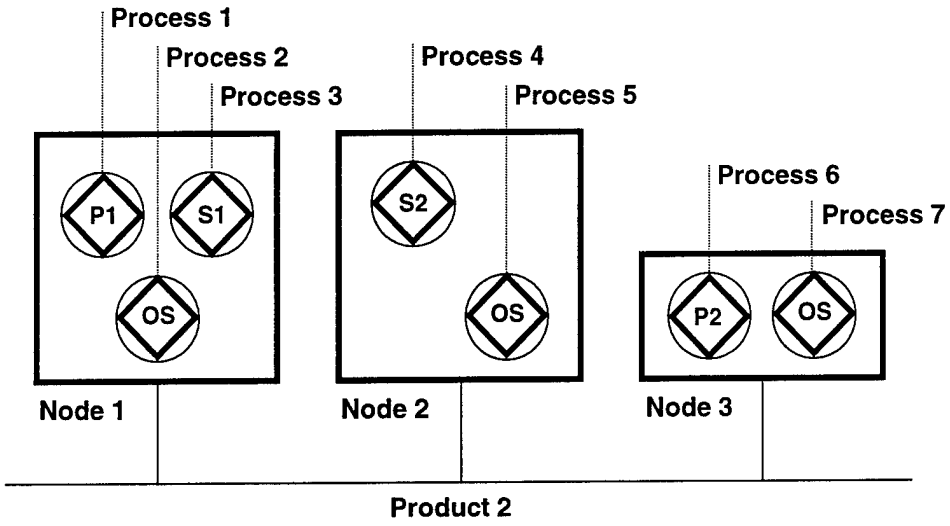
Figure 13: Alternative Node Structures

Figure 14 shows an allocation of the units of deployment onto the nodes in Node Structure 1 and Figure 15 shows an allocation onto the nodes in Node Structure 2.



Key	
P1	UI Page service 1
P2	UI Page service 2
S1	Service 1
S2	Service 2
OS	Operating system

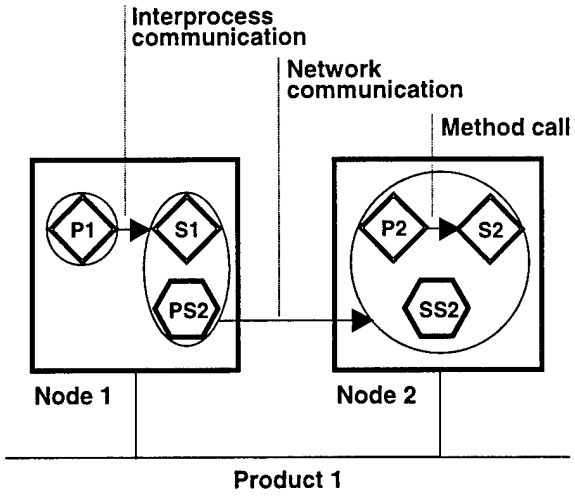
Figure 14: Distribution of the Units of Distribution and Processes for the Product Based on Node Structure 1



Key	
P1	UI Page service 1
P2	UI Page service 2
S1	Service 1
S2	Service 2
OS	Operating system

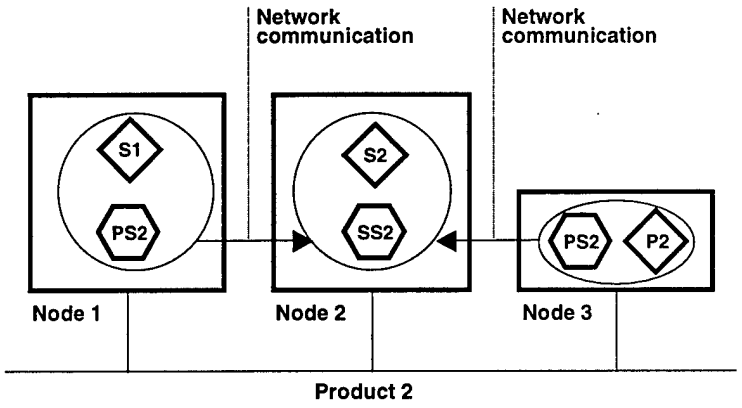
Figure 15: Distribution of the Units of Distribution and Processes for the Product Based on Node Structure 2

The next step is to understand how physically the Units of Distribution will communicate. In some cases (e.g., Unit of Deployment Service 1 uses functions from the Unit of Deployment Service 2) the communication mechanisms must be used to guarantee the cooperation of the Units of Deployment residing on a single Node. This is desirable so that the units of deployment do not need to be modified to reflect their current assignment to a node. We show this for Product 1 Figure 16 and for Product 2 in Figure 17.



Key	
P1	UI Page service 1
P2	UI Page service 2
S1	Service 1
S2	Service 2
PS2	Proxy service 2
SS2	Stub service 2

Figure 16: Communication Mechanisms for the Product Based on Node Structure 1



Key	
P2	UI Page service 2
S1	Service 1
S2	Service 2
OS	Operating system
PS2	Proxy Service 2
SS2	Stub Service 2

Figure 17: Communication Mechanisms for the Product Based on Node Structure 2

We have seen that the **Virtual Thread** *Command Service 1* is used both in the Unit of Deployment *Service 1* and in the Unit of Deployment *Service 2*. The Unit of Deployment *UI Page Service 2* and The Unit of Deployment *Service 2* must similarly cooperate, but only in the product based on the Node Structure 2. Because in both cases the two Units of Deployment reside on different nodes we introduced a mechanism to allow the information to flow through the network.

Two new types of threads have been introduced in order to allow the threads or, more precisely, the communication between threads, to cross the network boundary. These are the "proxy" and the "stub" threads. The proxy thread is started as soon as an attempt to give control to an external component is made. The proxy thread handles the network communication on the sender side. The stub thread receives the necessary information from the proxy thread and translates it back to the requested call at the component.

It's logical to assume that beneath this mechanism is also a mechanism to name and to retrieve objects in a distributed environment.

The proxy-stub mechanisms for the product base on the Node Structure 2 is described in Figure 18.

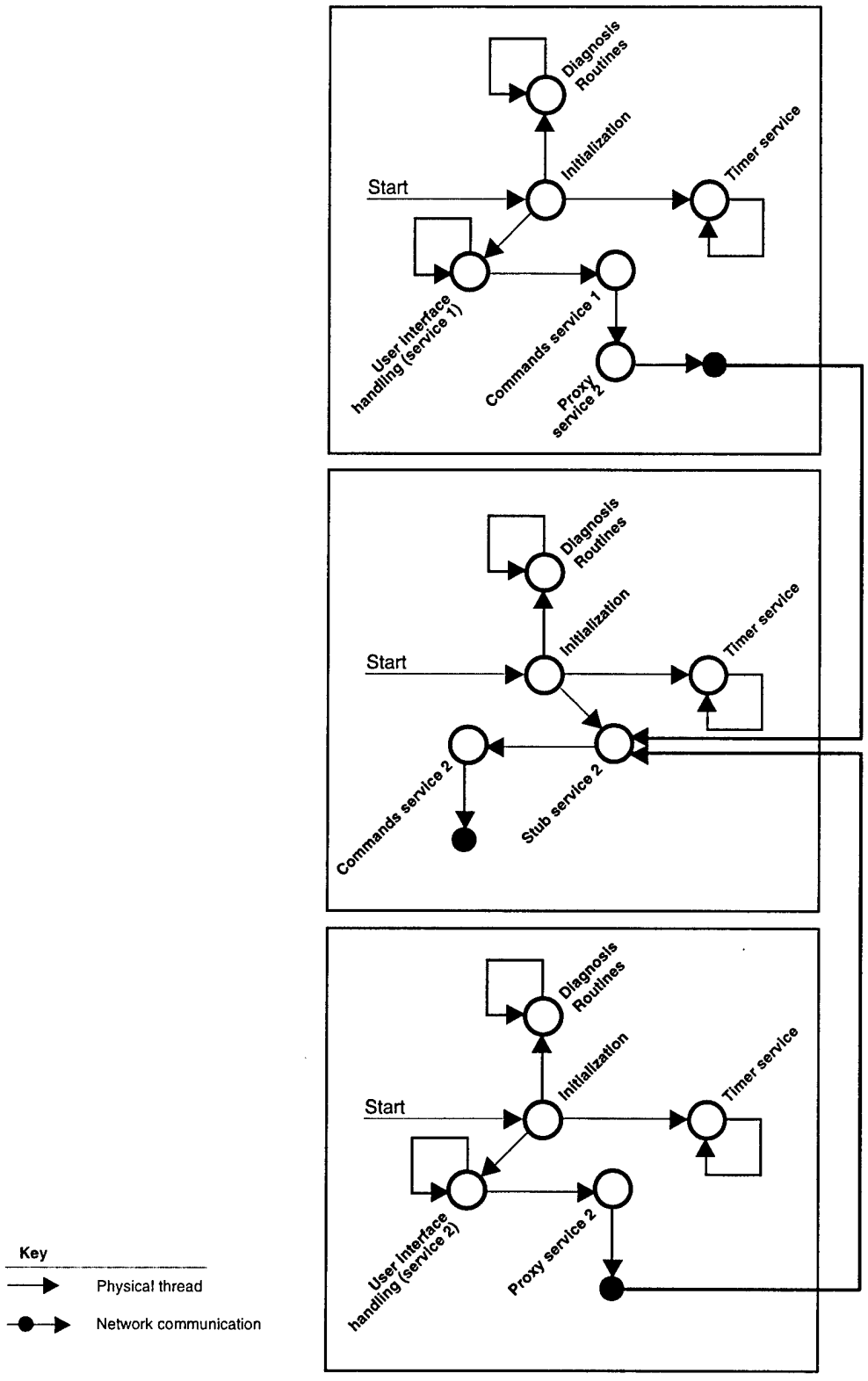
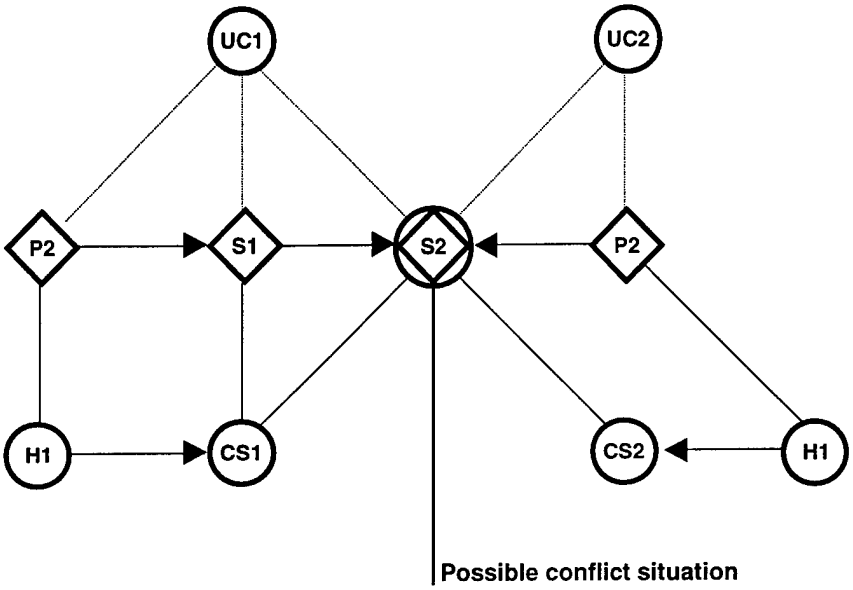


Figure 18 Virtual Threads vs. Physical Threads

In the last step we consider the possibility of conflicts between threads running in parallel. Assuming that the originating Use Cases applied to the communication between the Functional Structure are shown in Figure 19, and the level of concurrency shown is chosen (i.e., Threads *Command Service 1* and *Commands Service 2*), we are able to discover a possible conflict in the Unit of Deployment Service 2.



Key	
P1	UI Page service 1
P2	UI Page service 2
S1	Service 1
S2	Service 2
OS	Operating system
UC1	Use case 1
UC2	Use case 2
⟶	Threads of control

Figure 19: Use Case Mapping

The situation can get more complicated if the degree of parallelism of the system is even higher, because, for example, we can have more instances of the Use Cases running in parallel.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (LEAVE BLANK)	2. REPORT DATE January 2000	3. REPORT TYPE AND DATES COVERED Final
4. TITLE AND SUBTITLE The Architecture Based Design Method		5. FUNDING NUMBERS C — F19628-95-C-0003
6. AUTHOR(S) Felix Bachmann, Len Bass, Gary Chastek, Patrick Donohoe, Fabio Peruzzi		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213		8. PERFORMING ORGANIZATION REPORT NUMBER CMU/SEI-00-TR-001
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116		10. SPONSORING/MONITORING AGENCY REPORT NUMBER ESC-TR-00-001
11. SUPPLEMENTARY NOTES		
12.A DISTRIBUTION/AVAILABILITY STATEMENT Unclassified/Unlimited, DTIC, NTIS		12.B DISTRIBUTION CODE

13. ABSTRACT (MAXIMUM 200 WORDS)

This paper presents the Architecture Based Design (ABD) method for designing the high-level software architecture for a product line or long-lived system. Designing an architecture for a product line or long-lived system is difficult because detailed requirements are not known in advance. The ABD method fulfills functional, quality, and business requirements at al level of abstraction that allows for the necessary variation when producing specific products. Its application relies on an understanding of the architectural mechanisms used to achieve this fulfillment.

The method provides a series of steps for designing the conceptual software architecture. The conceptual software architecture provides organization of function, identification of synchronization points for independent threads of control, and allocation of function to processors. The method ends when commitments to classes, processes and operating system threads begin to be made. In addition, one output of the method is a collection of software templates that constrain the implementation of components of different types. The software templates include a description of how components interact with shared services and also include "citizenship" responsibilities for components.

14. SUBJECT TERMS Architecture-based design method, software architecture			15. NUMBER OF PAGES 56
16. PRICE CODE			
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102