Technical Report
CMU/SEI-94-TR-10
ESC-TR-94-010

Toward Deriving Software Architectures From Quality Attributes

Rick Kazman, University Of Waterloo

Len Bass, Software Engineering Institute

August 1994

Technical Report
CMU/SEI-94-TR-10
ESC-TR-94-010
August 1994

Toward Deriving Software Architectures From Quality Attributes

Rick Kazman, University Of Waterloo

Len Bass, Software Engineering Institute

Software Architecture Attribute Engineering Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the

SEI Joint Program Office
HQ ESC/AXS
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

(signature on file)

Thomas R. Miller, Lt Col, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

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.

This document is available through Research Access, Inc., 800 Vinial Street, Pittsburgh, PA 15212.
Phone: 1-800-685-6510. FAX: (412) 321-2994. RAI also maintains a World Wide Web home page. The URL is http://www.rai.com

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center, Attn: FDRA, Cameron Station, Alexandria, VA 22304-6145. Phone: (703) 274-7633.

Table of Contents

1 Software Quality Attributes 1

2 Non-Functional Qualities and Architecture 3

3 With Respect to What? 5

4 Operations for Realizing Quality Attributes 7
4.1 Separation 7
4.2 Abstraction 8
4.3 Compression 8
4.4 Uniform Composition 9
4.5 Replication 9
4.6 Resource Sharing 10
4.7 Unit Operations and Quality Attributes 10

5 Constructing Software Architectures: Case Studies 13
5.1 User Interface Management Systems 13
5.2 Compilers 17

6 Conclusions and Future Work 21

7 Acknowledgments 23

References 25

Appendix A Unit Operations and Quality Attributes 27
A.1 Scalability 28
A.2 Separation 28
A.3 Modifiability 28
A.4 Integrability 29
A.5 Portability 30
A.6 Performance 31
A.7 Reliability 32
A.8 Ease of Creation 33
A.9 Reusability 34

List of Figures

Figure 5-1: Creation of Software Architectures Based On Quality Attributes 13
Figure 5-2: Functional Representation of a System 14
Figure 5-3: Abstraction of the Presentation 14
Figure 5-4: Abstraction of the Dialogue 15
Figure 5-5: The Final UIMS Functional Partitioning 16
Figure 5-6: Application of Uniform Composition to Dialogue 17
Figure 5-7: Functional Decomposition of Compiler 18
Figure 5-8: Applying Abstraction to Functional Decomposition of Compiler 18
Figure 5-9: The Final Compiler Architecture 19

Toward Deriving Software Architectures From Quality Attributes

Abstract: A method for deriving software architectures from a consideration of the non-functional qualities of the system is presented. The method is based on identifying a set of six “unit operations” and using those operations to partition the functionality of the system. These unit operations were derived from the literature and from expert practice. The relationship between the unit operations and a set of eight non-functional qualities is explored. Evidence is provided for the validity of the method by using it to derive six well-known architectures from the areas of user interface software and compiler construction.

1 Software Quality Attributes

When creating a new software architecture for some application domain, system designers often justify their creation by claiming that it supports, and even promotes, certain qualities, often called *non-functional qualities*. These qualities—things like portability, reusability, performance, modifiability, and scalability [DOD 88]—are supposed to be automatically conferred on any system that is realized using the architecture.

However, there has been little research focussed on the question of precisely how it is that a software architecture realizes and promotes these qualities. Designers typically rely on their intuition and experience. They often appeal to ad hoc arguments to justify their design decisions. This doesn’t mean that their design decisions are wrong or that they routinely create inferior architectures; experienced designers regularly create good architectures. What it does mean is that theirs is a black art, a skilled craft, a knack learned only after years of hard-won experience. This view of design as an apprenticeship is common in software engineering.

The beliefs that underlie this paper are that

- The achievement of non-functional qualities of a system are intimately connected with the software architecture for that system.
- These qualities can be achieved through the appropriate application of a set of “unit operations.”[1]
- Specific architectures can be derived from an understanding of the unit operations and the non-functional qualities to be achieved by that architecture.

---

1. The term “unit operations” is chosen by analogy with Chemical Engineering [Shaw 90]. These are structure transforming operations that are ubiquitous in the design of software systems.

Note that we are saying that non-functional qualities influence the software architecture, not that non-functional qualities are achieved *exclusively* by architectural means. The achievement of non-functional qualities is attributable to many factors (such as coding styles, documentation, testing, etc.), but the larger the system, the more the achievement of non-functional qualities rests in a system's software architecture [Abowd 93].

Our objective in this paper is to provide evidence that these beliefs are well founded. We do this in two ways:

1. We argue that the achievement of non-functional qualities is the primary motivator for the architecture of large, complex systems.
2. We provide a set of candidate unit operations; show through case studies from well-understood, mature domains how these operations are manifested; and derive from "first principles" well-known architectures in user interfaces and compilers on the basis of these unit operations.

The unit operations that we propose are not new. They are operations used every day by experienced designers: separation, abstraction, composition, and so on. Our purpose is not to define new primitives for building systems or to propose a notation or language for describing architectures but to show how the operations realize software qualities. Our claim is that codifying derivations based on unit operations and their relationship with non-functional qualities will allow the creation of architectures to become a rote activity as it is in traditional engineering disciplines [Shaw 90], rather than an activity that is the special domain of wizards.

We are also interested in how qualities interact. Software qualities may be harmonic, may be mutually incompatible, or may exist in a state of mutual tension. For example, a system that attempts to maximize portability may sacrifice runtime performance; a system that is maximally scalable may achieve that quality at the cost of development efficiency. Although there is no single right way to resolve these conflicts, we can at least make the architectural tradeoffs explicit. This will allow a designer to make architectural decisions based on an explicit set of priorities—if portability is more important than performance, one sort of architecture results; if performance is of paramount importance, a different architecture is created. Such tradeoffs occur routinely in software development, but are seldom planned for, predicted, or explicitly recorded [Lindstrom 93].

2 Non-Functional Qualities and Architecture

In this section, we explore the relationship between the software architecture of a system and the non-functional qualities to be achieved by that system. We argue that there is an intimate connection between a system's architecture and the ability to achieve particular non-functional qualities within that system. Given the ambiguities in common usage of these terms, we first will say explicitly what we mean by both "software architecture" and "non-functional qualities."

The software architecture of a system is the module-level design of that system ([Garlan 93], [Perry 92])—in contrast to more traditional views of software that concentrate on comparatively low-level details: data structures and algorithms. Software architecture can be understood by viewing a system's modules in terms of their

- Functionality (and how that functionality is decomposed).
- Structure (its computational components and connectors, both data and control).
- Allocation of functionality to its structure [Kazman 94].
- Coordination model [Gelernter 92]

The non-functional qualities of the system are orthogonal to the functionality of the system. We view the functionality as the mapping of input to output generated by the system. This narrow view throws into the non-functional realm any discussion of performance, modifiability, portability, scalability, and so on.

Our assertion that the system architecture is intimately connected to the achievement of non-functional qualities should not be controversial. As systems grow in size and complexity, the achievement of non-functional qualities resides increasingly in architectural decisions.

Parnas [Parnas 72] in his seminal paper on competing allocations of function to structure gave as a canonical problem the creation of a KWIC (key word in context) index. Parnas presented two different software structures that solve the problem and analyzed these solutions in terms of their support for future modifications. Garlan and Shaw later extended Parnas' example to four different architectures and examined these architectures with respect to modifiability, performance, and reuse [Garlan 93]. The architectures presented were all capable of performing the functionality desired; the distinctions among them were based on their satisfaction of non-functional qualities.

The experience of expert software designers at the Software Engineering Institute has corroborated the view that non-functional qualities "live" in a system's software architecture.

3 With Respect to What?

One difficulty in discussing software qualities is that they do not exist in the abstract, and cannot be described in the abstract. They are manifested in software and hardware and they exist within an environment of user and institutional needs: tasks to be achieved and anticipated usage. Thus, when we speak about a quality such as modifiability, we can only discuss designing for *a specific set* of anticipated modifications. This will be based on our experience with similar systems and how these systems tend to grow and change over time. Similarly, when we speak about optimizing for performance, we always think of performance in terms of well-understood problem areas—database query optimization, file merging, semantic feedback in user interface systems, compiler code optimization.

As a consequence, when we analyze an architecture or want to build an architecture, we can't simply say "portability is important," or even "portability is most important." While we definitely need to make both types of decisions—determining the quality attributes of interest and prioritizing them—we must couch these decisions in terms of anticipated usage scenarios. In other words, we must create a set of benchmark tasks for each quality attribute that typify the modifications, efficiency requirements, or changes of platform that the architecture must support. This grounds our claims of scalability or reliability or portability in terms of actual anticipated needs.

4 Operations for Realizing Quality Attributes

To justify our belief in the existence of unit operations from which a software architecture can be derived, we will present six candidate unit operations—separation, abstraction, uniform composition, resource sharing, replication, and compression—and discuss them in terms of their effects on non-functional qualities. In particular, we are interested in:

- The unit operations necessary to achieve a chosen quality attribute in isolation.
- The justification for these operations.
- The effects on an architecture of combining quality attributes.

We now describe and explain the motivation for each unit operation. In Section 5 we will show, through case studies of compiler and user interface software, how desired non-functional qualities have been achieved through the application of these unit operations.[2]

4.1 Separation

Separation is a unit operation that places a distinct piece of functionality into a distinct component that has a well-defined interface to the rest of the world [Parnas 72]. The separation operation has been applied whenever an architecture contains more than one module. Separation isolates a portion of a system's functionality. The portion of a system's functionality that is isolated is determined by the desire to achieve certain quality factors.

For example, separation can be used to ensure that changes to the external environment do not affect a component, and changes to the component do not affect the environment, as long as the interface is unchanged. This is reflected in Table 1 by the + marks for portability and modifiability.

Common examples of separation are found in

- Data flow architectures [Garlan 93] (pipes and filters and batch sequential systems).
- Old-fashioned compilers [Aho 87] where each compilation phase—lexical analysis, syntactic analysis, code generation, etc.—was a separate process.
- User interface management systems [Pfaff 85], where presentation, dialogue and application concerns are separated.

2. Our choice of terms for unit operations is meant to suggest an operational view of existing structural patterns. For example, the term *separation* comes from the existing notion of *separation of concerns*.

4.2 Abstraction

Abstraction is the operation of creating a virtual machine. A virtual machine is a component whose function is to hide its underlying implementation. Virtual machines [Dijkstra 68] are often complex pieces of software to create, but once created can be adopted and reused by other software components, thus simplifying their creation and maintenance (because they reference a set of abstract functionality).

Virtual interfaces are found anywhere there is a need to emulate some piece of functionality that is non-native; for example, to simulate a parallel computation on a single processor. Another common use of virtual interfaces is in layered systems. For example, in the ISO OSI (International Organization for Standardization open systems interconnection) model, lower layers provide a virtual interface to higher layers—a generic, idealized functionality that hides implementation details. A third use is to provide a common interface to a heterogeneous set of underlying implementations.

For example, virtual toolkits are becoming quite common features of user interface management systems. If one wanted to run an application on platforms that supported X11, OpenLook, MS-Windows, and the Macintosh toolkit, one would likely define a virtual interface between the system's presentation and the rest of the functionality [Rochkind 92]. In this way, the user interface portion of the software is only written once, in terms of an abstract virtual toolkit for the user interface.

4.3 Compression

Compression is the operation of removing layers or interfaces that separate system functions, and so it is the opposite of separation. These layers may be software (process boundaries, procedure calls) or hardware (separate processors). When one compresses software, one takes distinct functions and places them together. The history of software engineering and computer science has tended away from compression: abstract data types, the client-server paradigm, distributed and parallel computing, and object-oriented development are all examples of the *addition* of layers (i.e. separation).

Compression serves two main purposes:

1. To improve system performance (by eliminating the overhead of traversing the layers between the functions). For example, Lindstrom discusses the use of layer elimination to meet performance goals in a fault-tolerant, real-time system [Lindstrom 93]. Other examples of compression for performance include semantic feedback in user interface systems and layer straddling in communication protocols.
2. To speed system development (by eliminating the requirement that different part of a system's functionality be placed in separate software components). For example, Microsoft's Visual Basic [Euler 91] allows a developer to directly couple individual user interface objects to application code.

A common technique for automatically achieving compression is the use of macros or inline procedures.

4.4 Uniform Composition

Composition is the operation of combining two or more system components into a larger component. *Uniform Composition* is a restriction of this operation, limiting the composition mechanisms to a small set. Having uniform composition mechanisms eases integration of components and scaling of the system as a whole.

For example the Model-View-Controller (MVC) paradigm [Krasner 88], the abstraction used in the Smalltalk language, decomposes a user interface into a set of uniform abstractions each of which contains a model (an application), a view (a presentation), and a controller (which maps between the two as well as between a hierarchical decomposition of MVC triples). Similar mechanisms have been used in the PAC (presentation abstraction control) paradigm [Coutaz 87].

Uniform composition has also been shown to be a powerful tool in the flight simulator domain for allowing large-scale systems to be composed from their subsystems and components, and for allowing them to be cleanly integrated [Abowd 93].

4.5 Replication

Replication is the operation of duplicating a component within an architecture. This technique is used to enhance reliability (fault tolerance) and performance. This unit operation is used in hardware as well as software. When components are replicated, it requires the simultaneous failure of more than one component to make the system as a whole fail. As the amount of replication in a system increases, the available work can be spread among more of the system's components, thus increasing throughput; however the chances of a single component failing increase dramatically.

The qualities of reliability and performance are in a mutual state of tension: reliability is increased through increased redundancy (i.e., having several components perform the same operation), whereas performance may be increased through increased parallelism (i.e., dividing a single function among several components). In the latter case, the failure of any of the parallel components may cause the entire operation to fail. Thus reliability and performance are in mutual tension. For example, disk array technology uses replication to enhance the performance of disk storage systems and to ensure that reliability is maintained (but it does not optimize either quality). Although a single disk drive will have a mean time to failure of 20-100 years, the mean time to failure of a non-redundant disk array may be on the order of only months or weeks [Ganger 94].

Examples of replication in software exist for both fault tolerance and performance. As an example of fault tolerance, the ISIS system [Birman 93] provides mechanisms for redundancy of both computation and communication. An example of replication for performance is the common use of data caching.

4.6 Resource Sharing

Resource sharing is an operation that encapsulates either data or services and shares them among multiple independent consumers. Typically there is a resource manager that provides the sole access to the resource. Shared resources, while they are often initially costly to build, enhance the integrability, portability, and modifiability of systems, primarily because they reduce the coupling among components.

Common examples of shared software resources are databases, blackboards, integrated computer-aided software engineering (CASE) tool environments, and servers (in a client-server system). In each of these cases, shared resources enhanced the integrability of the system.

For example, data repositories such as blackboards and databases are shared resources where the resource is persistent data to be stored and retrieved. Security kernels also manage shared resources, typically access to privileged system data and functionality. Integrated CASE environments rely on the notion of a “tool bus” or an explicit shared repository [Wasserman 89] to allow easy integration of tools.

4.7 Unit Operations and Quality Attributes

Given the above analyses, we now present a table of eight non-functional qualities and six unit operations. In each cell of the table, we indicate whether the desired quality is promoted by the unit operation (as indicated by a +) or inhibited by the unit operation (as indicated by a -). For cases in which the unit operation has no predictable effect on a quality attribute, we leave the cell empty or indicate both possible effects, depending on some other considerations.

Table 1: Quality Factors Versus Design Operations

Unit Operation	Software Quality Factor							
	Scalability	Modifiability	Integrability	Portability	Performance	Reliability	Ease of Creation	Reusability
Separation	+	+	+	+	+/-		+/-	+
Abstraction	+	+	+	+	-		+	+
Compression	-	-	-	-	+		+/-	-
Uniform Composition	+		+				+	
Replication	-	-		-	+/-	+	-	-
Resource Sharing		+	+	+	+/-	-	+	+/-

Each cell of this table is, in effect, a separate claim about the effect of a unit operation on a system's qualities in isolation. These claims are argued in the appendix. It should also be noted that not all unit operations are orthogonal. For example, resource sharing, uniform composition, and abstraction are all specializations of separation.

5 Constructing Software Architectures: Case Studies

To construct a software architecture that realizes some set of non-functional qualities, we must first understand the ramifications of each quality of interest. In particular, we must understand how the functional partitioning is affected by each quality attribute. As we have already said, this process is currently done intuitively by software developers.

We will now present a method for understanding these architectural decisions and hence for creating architectures. At an abstract level, the method consists of transforming non-functional requirements into allocations for each quality attribute in isolation, and then composing these individual allocations into a complete architecture, as shown in Figure 5-1:.

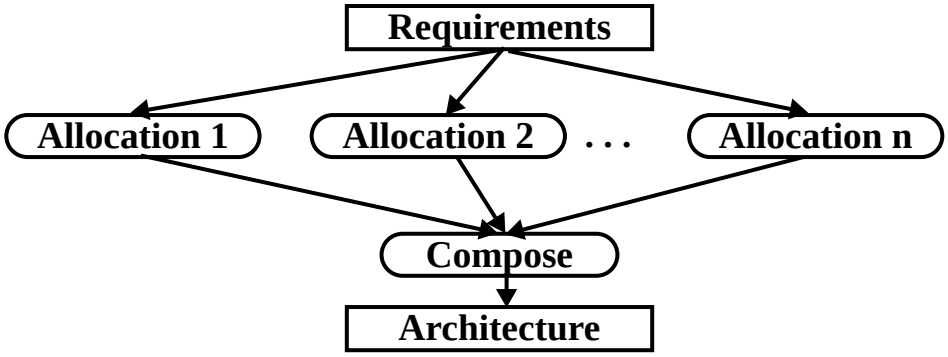

Figure 5-1: Creation of Software Architectures Based On Quality Attributes

We will exemplify the processes involved in this method (the tasks enclosed in ovals in Figure 5-1) in the following sections.

5.1 User Interface Management Systems

Since the user interface is frequently modified and user interfaces are often ported from toolkit to toolkit, user interface management systems (UIMSs) must support *modifiability* and *portability*.[3] The first steps in designing an architecture to meet these non-functional quality goals is to adopt a provisional functional analysis for the user interface system and analyze the nature of modifiability and portability in this domain.

There are at least two separate kinds of functions that any system with a user must provide: *presentation* (interaction with the user), and *application* (the underlying purpose of the system) [Pfaff 85]. It has been further noted that there are temporal and hierarchical aspects to human-computer interaction—the main task to be accomplished by the user (compose a document,

3. For example, this paper is being composed using a desktop publishing package that runs on Unix/Motif, MS Windows, and Macintosh platforms. The functionality is the same on each platform. Only the presentation differs.

create a spreadsheet, send mail) is broken down into sub-tasks (create a new file, modify a column, type a paragraph), which are further subdivided until we arrive at the level of physical actions (type a character, click on an icon, pull down a menu) that the user performs on the presentation. This decomposition and sequencing can be thought of as a dialogue between the user and the application. Thus *dialogue* is a third type of function that every interactive system must support.

We say nothing at this point about how these functions are partitioned. One can think of a purely functional representation of a system as having the structure shown in Figure 5-2 below.[4] The functions are lumped together in a single structural component. However, given that we want to design our system with modifiability and portability in mind, we will invoke the unit operations of separation and abstraction. We now turn to this task.

Figure 5-2: Functional Representation of a System

As discussed in Section 3, non-functional qualities are abstract; they are conceptual categories. To make them meaningful for a particular application domain, they must be reified as particular tasks [Kazman 94]. In the user interface domain, we can identify two types of portability concerns: replacing the presentation toolkit (which is quite common) and replacing the application (which is rather more rare).

To plan for portability, we want to mitigate the effects of replacing the presentation toolkit. Thus, we use the abstraction operation to isolate the presentation in its own component (typically a shared resource). At this point, our functional partitioning looks like this:

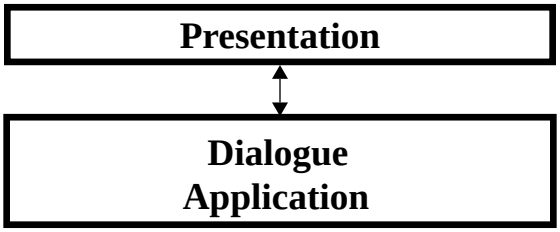

Figure 5-3: Abstraction of the Presentation

4. In this diagram and those that follow, a box indicates a collection of functionality and an arrow indicates an association among functions, such as data flow.

To insulate against modifications to the dialogue (typically the most heavily modified portion of an interactive system), or modifications to the application, we once again appeal to the operation of abstraction: we isolate the dialogue into a separate component. Our functional partitioning now looks like this:

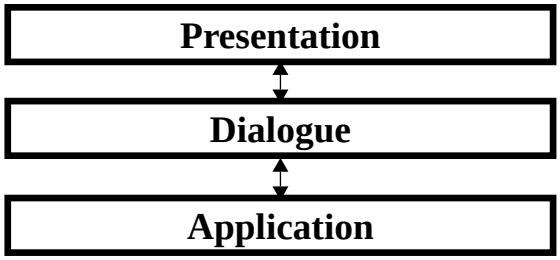

Figure 5-4: Abstraction of the Dialogue

5.1.1 Seeheim

We have now derived the basis of the Seeheim model of UIMSs [Pfaff 85] from the application of unit operations. The derivation of well-known software architectures from unit operations is our validation of the construction technique presented in Figure 5-1. By linking non-functional requirements to unit operations, we have derived the same architectures as those derived by experienced designers.

This does not imply that the above model is ideal, however. Implementations of the Seeheim model have well-documented problems. For example, given that the presentation and dialogue have been separated, there arises the possibility of their interaction. When one replaces the presentation toolkit, one should not have to rewrite the dialogue to use the idiosyncratic objects and attributes of the new toolkit. Similarly, if one modifies the dialogue, one does not want to have to maintain several related modifications, one per toolkit used.

Thus, to mitigate the interaction of portability and modifiability (changing the toolkit and modifying the dialogue), we appeal to another unit operation: abstraction. We make the connection between the presentation and dialogue components indirect. To accomplish this, we insert a function between the presentation and dialogue that maps between the two, manifesting a virtual presentation toolkit to the dialogue, thus forcing the dialogue to conform to the abstractions presented by the virtual toolkit.

Finally, we can consider another type of potential modification. If the application is replaced frequently (for example in an interface to a multi-database), one would once again appeal to the mechanism of abstraction, inserting a component between the application and the dialogue that maps between the two in the same way that a virtual user interface toolkit maps between the presentation and dialogue. We have called this function a *virtual application*, on analogy with a virtual toolkit. These layers buffer the dialogue from changes in the operating environment.

The final functional partitioning engendered by our set of benchmark tasks and our primitive mechanisms for modifiability is shown in Figure 5-5.

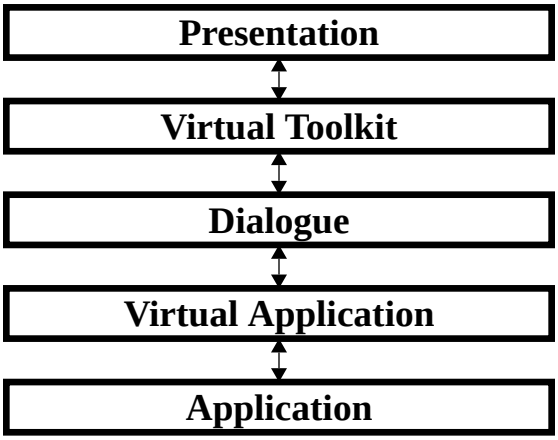

Figure 5-5: The Final UIMS Functional Partitioning

5.1.2 Arch/Slinky

We have now derived the Arch/Slinky model of user interface management systems [UIMS 92] entirely from first principles. Each of these models, Seeheim and Arch/Slinky, were created by a group of experts in the field of user interface software. What we have done is given a method whereby even a non-expert can derive such models by the application of architectural unit operations. Note that what we have in Figure 5-5 is not yet a software architecture; it is a functional partitioning with an indication of data flow. We have not specified the software structure into which this functional partitioning maps.

This functional partitioning suggests an architecture that strongly supports the qualities of modifiability and portability. The degree to which the software structure maintains the functional partitioning will affect the eventual success of the architecture with respect to these qualities [Kazman 94].

5.1.3 PAC

Up until now we have been presenting and rationalizing our functional decomposition with respect to modifiability and portability. What happens if we add another non-functional quality requirement to the mix? For example, given that the dialogue component in an interactive system is large, complex, and continually modified, we would like that component to support the quality of scalability. Appealing to Table 1, we can see that the quality of scalability is addressed by the unit operations of separation and uniform composition. From the descriptions of each unit operation in sections 4.1 through 4.6, we can see that uniform composition is a refinement of separation, and so is a stronger operation than separation.

Thus uniform composition is the most applicable unit operation for ensuring scalability of a single component. We will apply this operation to the dialogue, thus allowing the dialogue to be decomposed into manageable, codable chunks, each of which can then be re-integrated to

the whole through a regular composition mechanism [Krasner 88]. A regular hierarchical composition mechanism is one such example. By adding uniform composition to the dialogue component, we derive a functional decomposition as follows:

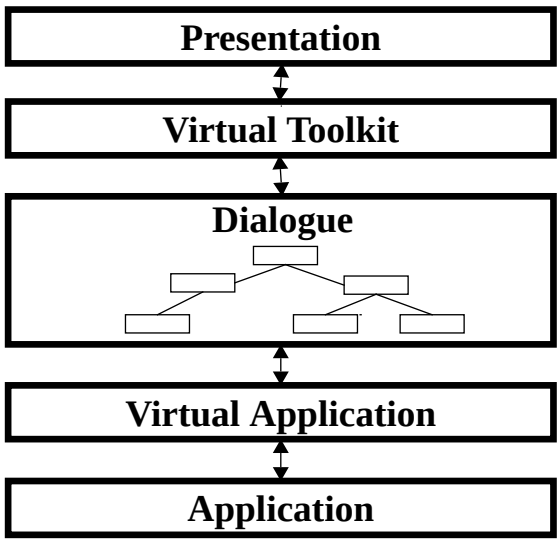

Figure 5-6: Application of Uniform Composition to Dialogue

We have now arrived at the PAC-AMODEUS model of user interface software [Nigay 91] from the application of unit operations. This model, the result of several years of research and experience, was explicitly designed to promote modifiability, portability, and scalability of the dialogue component.

5.2 Compilers

We now turn to a historical examination of the software architectures of compilers in terms of how these architectures address non-functional requirements. Or, to put it another way, we will reconstruct the history of compiler development as driven by non-functional requirements and their realization in unit operations. We use this example because compiler software architecture is well understood and extensively studied ([Aho 87], [Garlan 93]), but the evolution of compiler software architecture has not been studied as a reaction to non-functional requirements.

Early compilers performed two functions: analysis of the source language and synthesis of machine code. As compiler construction became relatively well understood, the analysis portion was subdivided into the following functions: lexical analysis, syntactic analysis, and semantic analysis. The synthesis portion typically consisted of code generation and optimization. These functions (or phases, as they're often called) had to be constructed and modified independently of each other. Thus, to promote the non-functional quality of modifiability, the unit operation of abstraction must be applied.

What results is a functional decomposition typical of the 1970s state of the art in compiler construction:

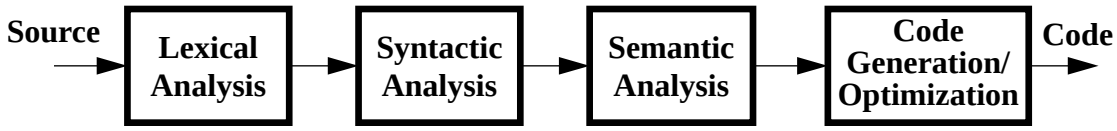

Figure 5-7: Functional Decomposition of Compiler

Each phase is separate and runs to completion before the next phase starts. Results are passed from phase to phase through intermediate files.

However, compilers are complex and costly pieces of software to create. Thus, it is natural that an organization creating a compiler would want to amortize the investment in creating a compiler over many different products (compilers for different hardware platforms). This means that the compiler must be portable. Once again, appealing to Table 1 and the following sections we can see that the unit operation of abstraction is most appropriate to apply here to engender portability. Essentially we would like the interface between the analysis and synthesis functions to be indirect. To realize abstraction, we insert a function between semantic analysis and code generation/optimization that takes the results of semantic analysis and translates this into generic machine code. The generic code can then be optimized and translated into machine-specific code, resulting in the functional decomposition shown below.

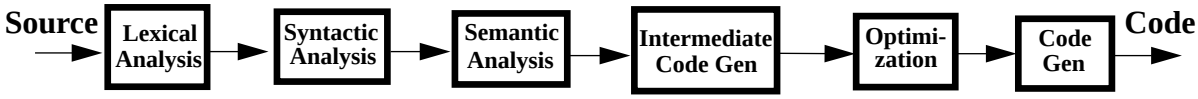

Figure 5-8: Applying Abstraction to Functional Decomposition of Compiler

The addition of the non-functional requirement of portability resulted in a functional decomposition that represented the state of the practice in compiler construction by the mid-1980s. Throughout the 1980s, however, many of the changes in compiler development resulted from the need to turn a compiler from a stand-alone tool into the heart of a suite of language-directed tools. Thus, the non-functional quality attribute of integrability became important to compiler software architecture.

To deal with this requirement, we can appeal to the unit operation of resource sharing. In particular, it is necessary to share the attributed parse tree and symbol table that the compiler creates and annotates during its various phases. This results in a very different software architecture, centered around the shared resources, as shown in Figure 5-9 below (adapted from [Garlan 93]).[5] This architecture now supports additional tools, such as structured editors and code analysis tools.

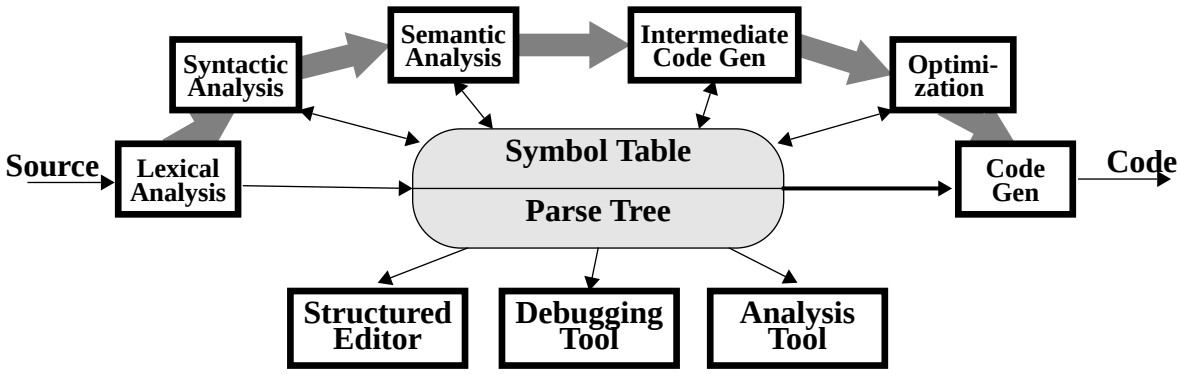

Figure 5-9: The Final Compiler Architecture

5. In Figure 5-9 the thin arrows represent data flow and the fat, grey arrows represent control flow. What this means is that all communication between the phases is through the shared symbol-table/parse tree resource.

6 Conclusions and Future Work

In this paper we have given the first steps for a process for creating software architectures from first principles. The principles, unit operations, are used for the achievement of non-functional requirements. Clearly some architectural decisions are necessitated by functional requirements, in which case the unit operations do not apply. However we have argued that, as systems grow, non-functional requirements tend to dominate architectural decisions. Consequently the software engineering field must find ways of addressing these requirements in a predictable, repeatable fashion.

The interest of the unit operations and the architecture construction method given in Section 4 is that it gives developers a set of standard techniques for constructing software architectures. We have also provided rationale for the use and application of each technique. Perry and Wolf [Perry 92] have stressed the importance of connecting rationale with architectural decisions, and we concur with this view.

In the future we can see several directions in which this work should be extended. We need to have a way to connect architectures in general, and unit operations specifically, to programming language issues (such as deferred binding and constraints). These language features greatly affect the achievability of many unit operations. We would also like to extend the set of unit operations and discuss their composition in more detail, addressing the tradeoffs that may be necessary when non-functional requirements are in direct conflict with each other. Finally, we would like to extend our analysis techniques to larger and less well-understood software domains.

We do not believe that the six unit operations that we have discussed represent a complete set. Obviously, we can't derive all architectures from this set. However we believe that this work, and in particular the method that underlies the use of unit operations, shows promise in making the achievement of non-functional properties predictable. Our confidence in this method is based on the fact that we were able to derive six different software architectures (three each for compilers and UIMSs) that represent the work and insight of expert developers over many years.

7 Acknowledgments

We would like to thank Joe Batman, Reed Little, and Tom Ralya for sharing with us some of their experience in software design. We would also like to thank Alan Brown, Paul Clements, Peter Feiler, Gregory Abowd, and Nelson Weiderman for their helpful comments on an earlier draft of this paper.

References

[Abowd 93] Abowd, G.; Bass, L.; Howard, L.; & Northrop, L. *Structural Modeling: an Application Framework and Development Process for Flighty Simulators* (CMU-SEI-93-14, ADA271348). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1993.

[Aho 87] Aho, A.; Sethi, R.; & Ullman, J. *Compilers: Principles, Techniques, and Tools*, Reading, MA: Addison-Wesley, 1987.

[Birman 93] Birman, K. "The Process Group Approach to Reliable Distributed Computing."*Communications of the ACM 36*, 12 (December 1993): 37-53.

[Coutaz 87] Coutaz, J. "PAC, An Implementation Model for Dialog Design," pp. 431-436. *Proceedings of Interact '87*. Stuttgart, Germany: September, 1987.

[Crispen 93] Crispen, R.; Freemon, B.; King, K.; & Tucker, V., "DARTS: A Domain Architecture for Reuse in Training Systems," pp. 659-668. *15th /ITSEC Proceedings*. San Antonio, TX: November 1993.

[Dijkstra 68] Dijkstra, E.W. "The Structure of the 'THE' Multiprogramming System." *Communications of the ACM 11*, 5 (May 1968): 341-346.

[DOD 88] United States Department of Defense. *Military Standard, Defense System Software Development* (DOD-STD-2167A). Washington, DC: United States Department of Defense, 1988.

[Euler 91] Euler, L.; Maffei, E.; & Rauch, A. "Create Real Windows Applications in a Graphical Environment Using Microsoft Visual Basic." *Microsoft Systems Journal 6*, 4 (July 1991): 57-70, 116.

[Ganger 94] Ganger, G.; Worthington, B.; Hou, R.; & Patt, Y. "Disk Arrays: High-Performance, High-Reliability Storage Systems." *IEEE Computer 27*, 3 (March 1994): 30-36.

[Garlan 93] Garlan, D. & Shaw, M. "An Introduction to Software Architecture," 1-39. Ambriola, V. & Tortora, G (eds.), *Advances in Software Engineering and Knowledge Engineering*, Volume I. Singapore: World Scientific Publishing, 1993.

[Gelernter 92] Gelernter, D. & Carriero, N. "Coordination Languages and their Significance." *Communications of the ACM 55*, 2 (February 1992): 97-107.

[Kazman 94] Kazman, R.; Bass, L.; Abowd, G.; & Webb, S.M. "SAAM: A Method for Analyzing the Properties of Software Architectures," pp. 81-90. *Proceedings of ICSE 16*. Sorrento, Italy: May 1994.

[Krasner 88] Krasner, G. & Pope, S. "A Cookbook for Using Model-View-Controller User Interface Paradigm in Smalltalk-80." *Journal of Object Oriented Programming* (August/September 1988): 26-49.

[Lindstrom 93] Lindstrom, D. "Five Ways to Destroy a Development Project." *IEEE Soft-*

ware 10, 5 (September 1993): 55-58.

[Nigay 91] Nigay, L. & Coutaz, J. “Building User Interfaces: Organizing Software Agents.” *ESPRIT ‘91 Conference*. Brussels, Belgium: November 1991.

[Parnas 72] Parnas, D. “On the Criteria to Be Used in Decomposing Systems into Modules.” *Communications of the ACM* 15, 12 (December 1972): 1053-1058.

[Perry 92] Perry, D. & Wolf, A. “Foundations for the Study of Software Architecture.” *SIGSOFT Software Engineering Notes 17*, 4 (October 1992): 40-52.

[Pfaff 85] Pfaff, G. (ed.). *User Interface Management Systems*. New York: Springer-Verlag, 1985.

[Rochkind 92] Rochkind, M.J., “An Extensible Virtual Toolkit (XVT) for Portable GUI Applications,” pp. 485-494*. Digest of Papers, COMPCON* (Spring 1992). San Francisco, CA: Thirty-Seventh IEEE Computer Society International Conference, February 1992.

[Shaw 90] Shaw, M. *Prospects for an Engineering Discipline of Software* (CMU-CS-90-165). Pittsburgh, PA: Carnegie Mellon University School of Computer Science, 1990.

[UIMS 92] UIMS Tool Developers Workshop. “A Metamodel for the Runtime Architecture of an Interactive System.” *SIGCHI Bulletin 24*, 1 (January 1992): 32-37.

[Wasserman 89] Wasserman, A. “Tool Integration in Software Engineering Environments,” pp. 137-149. Long, F. (ed.)*, Software Engineering Environments*, Lecture Notes in Computer Science No. 467. Berlin, Germany: Springer-Verlag, 1989.

Appendix A Unit Operations and Quality Attributes

This appendix provides arguments for the table of the effects of unit operations on software quality factors. Below is the table presented in Section 4.7 that presents the relationships. Each cell of the table is a separate claim about the relationships between unit operations and quality attributes, and we expand on these claims here. We are interested, primarily, in most significant relationships between the quality factors and the unit operations. Thus, although there are many second-order relationships, we leave cells of the table blank when only second-order relationships exist. Furthermore, there may be mitigating factors in these relationships, but we do not discuss them because they are typically minor effects. For example, separation may have a positive effect or a negative effect on portability, depending on whether the separation encloses and hides system dependencies or whether it cuts across them. In our discussion of the quality attributes, we assume that unit operations are used in a way that is consistent with their major effects.

Table 2: Quality Factors Versus Design Operations

Unit Operation	Software Quality Factor							
	Scalability	Modifiability	Integrability	Portability	Performance	Reliability	Ease of Creation	Reusability
Separation	+	+	+	+	+/-		+/-	+
Abstraction	+	+	+	+	-		+	+
Compression	-	-	-	-	+		+/-	-
Uniform Composition	+		+				+	
Replication	-	-		-	+/-	+	-	-
Resource Sharing		+	+	+	+/-	-	+	+/-

Just as quality attributes do not exist in the abstract, these relationships do not exist in the abstract. That is, modifiability cannot be accurately discussed without reference to specific types of modifications to be performed. Abstraction as a unit operation to support modifiability assumes that

- There is a particular set of modifications under discussion.
- The portions of the functionality of the system that are being separated and abstracted are loosely coupled.

The assumption of specificity with respect to the quality attributes and the unit operations is made in each of the sections below, and so we don't repeat it in each section.

We organize this discussion by the quality attributes.

A.1 Scalability

Scalability is the ability of a system to support modifications that dramatically increase the size of the system.

A.2 Separation

To have the ability to dramatically increase the size of a system and maintain intellectual control, the additions must be broken into smaller pieces, and these smaller pieces must have specific interfaces with each other and with the original portion of the system. The existence of interfaces is the hallmark of separation. Hence, a + in the table.

A.2.1 Abstraction

The use of abstractions promotes the ability to construct large systems. This is the rationale, for example, for higher order language generations. Few people build large systems in assembler any longer. They build large systems on top of more abstract machines. Hence, a + in the table.

A.2.2 Compression

This is the reverse argument from that presented under separation. Very large systems cannot be constructed without some form of separation. Thus, every removal of an interface makes the system less easy to understand and inhibits scalability. Hence, a - in the table.

A.2.3 Uniform Composition

Very large systems are most easily constructed by having a small set of parts and combining those parts in a similar fashion. This reduces the intellectual effort required to understand the parts and to understand their interaction. The operation of uniform composition states that the combination of parts should occur in a regular fashion. Hence, a + in the table.

A.2.4 Replication

Having multiple copies of either software components or functionality of a very large system makes the system even larger and more difficult to comprehend. Hence, a - in the table.

A.2.5 Resource Sharing

Very large systems may have shared resources and can exist without shared resources. Hence, no entry in the table.

A.3 Modifiability

Modifiability is the ability of a system to be extended to accomplish additional functionality.

A.3.1 Separation

The creation of additional interfaces will enhance modifiability if it separates distinct pieces of functionality into distinct software components. This is a rationale for object-oriented programming and procedural abstraction. Hence, a + in the table.

A.3.2 Abstraction

The use of virtual machines is a technique that hides unnecessary detail and makes the addition of functionality easier to achieve as long as the new functionality

- is hidden in one of the virtual machines;
- can be achieved by composition of existing virtual machines; or
- can be achieved by the creation of a new virtual machine.

This is a common technique for supporting modifiability. Hence, a + in the table.

A.3.3 Compression

The removal of interfaces makes modifications more likely to have side effects. Hence, a - in the table.

A.3.4 Uniform Composition

The fact that components may be composed in a systematic fashion does not pertain to the ways in which functionality is divided among these components. Division of functionality is the key to modifiability. Hence, no entry in the table.

A.3.5 Replication

Any modification that involves either replicated data or functionality must be performed twice, once for each replicate. Hence a - in the table.

A.3.6 Resource Sharing

Modifications to data or functionality that is shared by multiple different consumers need only be made once. This is the reverse of the argument used in replication. Hence, a + in the table.

A.4 Integrability

Integrability is the ability to easily integrate separate systems or components of a system.

A.4.1 Separation

The components to be integrated are, by definition, already separate. However, if these components are broken into rational pieces of functionality, the integration of the components is simplified because internal dependencies will be easier to disentangle. Hence, a + in the table.

A.4.2 Abstraction

Abstraction is a means of achieving a common set of integration mechanisms. Hence, the problem of integration is localized to the support of these common mechanisms. We reflect this with a + in the table.

A.4.3 Compression

Integration of independently developed components is simplified by separating and exposing the elements of those components that must communicate. Insofar as compression removes this separation, it makes integration more difficult. Hence, a - in the table.

A.4.4 Uniform Composition

When the mechanisms for composing components are identical, it becomes easier to integrate those components. Hence, a + in the table.

A.4.5 Replication

When functionality or data is replicated, integrating it with the remainder of the system adds some difficulty. However, usually this effect is minor. Hence, no entry in the table.

A.4.6 Resource Sharing

A shared resource such as a shared database can become the integrating mechanism for a system. New components can be integrated by defining how they interact with the shared resource. In this case, integration is simplified by the existence of a shared resource. This is quite common in CASE tool environments, for example. Hence, a + in the table.

A.5 Portability

Portability is the ability of a system to execute on different hardware and software platforms.

A.5.1 Separation

Separation, if it encloses and hides platform dependencies, will help achieve portability. Hence, a + in the table.

A.5.2 Abstraction

The use of virtual machines is a technique that hides unnecessary detail and makes portability easier to achieve by hiding platform dependencies. Hence, a + in the table.

A.5.3 Compression

The removal of interfaces will merge platform-dependent and platform-independent implementations. This will make moving among hardware and software platforms more difficult. Hence, a - in the table.

A.5.4 Uniform Composition

The fact that components may be composed in a systematic fashion does not pertain to the ways in which functionality is divided among these components. Localizing platform dependencies is the key to portability. Hence, no entry in the table.

A.5.5 Replication

Any platform dependencies that involve either replicated data or functionality must be modified twice to execute on another platform, once for each replicate. Hence, a - in the table.

A.5.6 Resource Sharing

Modifications for different platforms to data or functionality that is shared by multiple different consumers need only be made once. Hence, a + in the table.

A.6 Performance

Performance is the measure of how well the computer system responds to its inputs. Common measures are response time, resource utilization, and throughput.

A.6.1 Separation

Separation requires the creation of additional interfaces and, in this case, hinders performance. On the other hand, parallelism is a technique used to improve performance, and the achievement of parallelism requires separation. In this case, separation supports performance. Hence, a +/- in the table.

A.6.2 Abstraction

The use of a virtual machine typically hinders performance because of the additional interface created and its associated data marshaling. Hence, a - in the table.

A.6.3 Compression

The removal of interfaces helps performance by removing the overhead of context switching. Hence, a + in the table.

A.6.4 Uniform Composition

The ability to easily compose components of a system has no effect on the runtime performance of that system. Hence, no entry in the table.

A.6.5 Replication

Replicating functionality or data can have both positive and negative effects on performance. When functionality or data is replicated, any computation must either be performed on all replicates or the results of any computation must be propagated to all replicates. In either case, this is a negative effect. On the other hand, replication is a technique used to support parallelism, and parallelism enhances performance. Hence, a +/- entry in the table.

A.6.6 Resource Sharing

Shared resources inhibit performance because access control to the resource adds additional computational load. On the other hand, optimization techniques can be applied to the shared resource and the performance of the system increased. Hence, a +/- entry in the table.

A.7 Reliability

Reliability is the ability of the system to sustain operations. A common measure is mean time between failures.

A.7.1 Separation

The division of functionality into several portions has both positive and negative impact on reliability. The positive impact is that since functionality is organized into smaller pieces, testing and the subsequent improvement of reliability becomes easier. The negative impact is caused by the additional interactions among components that occur when there are more components. In either of these cases, the impact is minor. Hence, no entry in the table.

A.7.2 Abstraction

The creation of abstractions has no significant effect on reliability. Hence, no entry in the table.

A.7.3 Compression

The removal of interfaces has the opposite impact from separation. Since the effect of separation is minor, the effect of compression is also minor. Hence, no entry in the table.

A.7.4 Uniform Composition

The easy composition of components has no effect on the reliability of the resulting system. Hence, no entry in the table.

A.7.5 Replication

Repeating functionality is one technique used to increase the reliability of systems because it allows for some number of component failures before the system as a whole fails. Hence, a + in the table.

A.7.6 Resource Sharing

The sharing of resources creates larger dependencies on the reliability of the resource being shared. The shared resource is a potential single point of failure. This is a problem for the reliability of the system as a whole. Hence, a - in the table.

A.8 Ease of Creation

Ease of creation is the difficulty of constructing the system. This is often measured in labor hours.

A.8.1 Separation

Since additional interfaces require additional work, separating functionality in simple systems increases the difficulty of creating a system. For complex systems, however, separation is a necessity for understanding. Hence, a +/- in the table.

A.8.2 Abstraction

Creation of a virtual machine increases the possibility for sharing the resulting abstraction. Assuming that the virtual machine is reused, abstraction will decrease the difficulty of creating a system, since the virtual machine functionality must be implemented only once. Hence, a + in the table.

A.8.3 Compression

Removal of interfaces reduces constraints on the developer of small systems and increases the complexity of interactions in large systems. This is the reverse of the argument for separation. Hence, a +/- in the table.

A.8.4 Uniform Composition

Having few mechanisms for composing components means that the developers need to have mastered fewer mechanisms, and this simplifies the construction process. Hence, a + in the table.

A.8.5 Replication

Replicating either functionality or data means that the developer must develop portions of the system twice and be concerned with integrating these multiple portions. Hence, a - in the table.

A.8.6 Resource Sharing

Creation of a shared resource decreases the difficulty of creating a system since some functionality must be implemented only once. Hence, a + in the table.

A.9 Reusability

Reusability is the reuse of existing code in a current development. This is only one of a variety of types of reuse, but it is the one most connected with the architecture of the system under development.

A.9.1 Separation

The less a component does and the more constrained its relationship with its environment, the higher the likelihood that it can be used in a subsequent development. Therefore, dividing functionality increases the possibility of reuse. Hence, a + in the table.

A.9.2 Abstraction

Creating a virtual machine means that the components that use the virtual machine can have a higher level of ignorance of their environment and can be implemented more quickly. Hence, a + in the table.

A.9.3 Compression

Removing interfaces increases the functional dependencies of a component with its environment and thus decreases the likelihood that it can be used in a subsequent development. Hence, a - in the table.

A.9.4 Uniform Composition

The ability to easily compose components is not one of the factors that determines whether the components themselves can be reused. Hence, no entry in this table.

A.9.5 Replication

Replicating functionality or data increases the constraints under which a component can be used and decreases the likelihood of reusing that component. Hence, a - in the table.

A.9.6 Resource Sharing

Sharing of resources increases the constraints on the components that use that resource. This inhibits the reuse of these components. On the other hand, it requires that the manager of the resource be usable by multiple consumers and this increases the likelihood that the resource manager can be reused. Hence, a +/- in the table.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION
Unclassified

1b. RESTRICTIVE MARKINGS
None

2a. SECURITY CLASSIFICATION AUTHORITY
N/A

3. DISTRIBUTION/AVAILABILITY OF REPORT
Approved for Public Release
Distribution Unlimited

2b. DECLASSIFICATION/DOWNGRADING SCHEDULE
N/A

4. PERFORMING ORGANIZATION REPORT NUMBER(S)
CMU/SEI-94-TR-10

5. MONITORING ORGANIZATION REPORT NUMBER(S)
ESC-TR-94-010

6a. NAME OF PERFORMING ORGANIZATION
Software Engineering Institute

6b. OFFICE SYMBOL (if applicable)
SEI

7a. NAME OF MONITORING ORGANIZATION
SEI Joint Program Office

6c. ADDRESS (city, state, and zip code)
Carnegie Mellon University
Pittsburgh PA 15213

7b. ADDRESS (city, state, and zip code)
HQ ESC/ENS
5 Eglin Street
Hanscom AFB, MA 01731-2116

8a. NAME OFFUNDING/SPONSORING ORGANIZATION
SEI Joint Program Office

8b. OFFICE SYMBOL (if applicable)
ESC/ENS

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
F1962890C0003

8c. ADDRESS (city, state, and zip code))
Carnegie Mellon University
Pittsburgh PA 15213

10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO	PROJECT NO.	TASK NO	WORK UNIT NO.
63756E	N/A	N/A	N/A

11. TITLE (Include Security Classification)
Toward Deriving Software Architectures From Quality Attributes

12. PERSONAL AUTHOR(S)
Rick Kazman and Len Bass

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (year, month, day)	15. PAGE COUNT
Final	FROM TO	August 1994	34 pp.

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (continue on reverse of necessary and identify by block number)
FIELD	GROUP	SUB. GR.	software analysis methods
			software architecture
			software construction

19. ABSTRACT (continue on reverse if necessary and identify by block number)

A method for deriving software architectures from a consideration of the non-functional qualities of the system is presented. The method is based on identifying a set of six “unit operations” and using those operations to partition the functionality of the system. These unit operations were derived from the literature and from expert practice. The relationship between the unit operations and a set of eight non-functional qualities is explored. Evidence is provided for the validity of the method by using it to derive six well-known architectures from the areas of user interface software and compiler construction.

(please turn over)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT
UNCLASSIFIED/UNLIMITED ■ SAME AS RPT □ DTIC USERS ■

21. ABSTRACT SECURITY CLASSIFICATION
Unclassified, Unlimited Distribution

22a. NAME OF RESPONSIBLE INDIVIDUAL
Thomas R. Miller, Lt Col, USAF

22b. TELEPHONE NUMBER (include area code)
(412) 268-7631

22c. OFFICE SYMBOL
ESC/ENS (SEI)

ABSTRACT — continued from page one, block 19