Software Engineering Institute

Attribute-Driven Design (ADD), Version 2.0

Rob Wojcik
Felix Bachmann
Len Bass
Paul Clements
Paulo Merson
Robert Nord
Bill Wood

November 2006

TECHNICAL REPORT
CMU/SEI-2006-TR-023
ESC-TR-2006-023

Software Architecture Technology Initiative
Unlimited distribution subject to the copyright.

Carnegie Mellon

This report was prepared for the

SEI Administrative Agent
ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2100

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.

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 2006 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 FA8721-05-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 252.227-7013.

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

Abstract vii

Acknowledgements ix

1 Introduction 1

2 ADD Overview 3

3 ADD Inputs and Outputs 7
- 3.1 Inputs to ADD 7
- 3.2 Outputs to Expect from ADD 9

4 Step 1: Confirm There Is Sufficient Requirements Information 11
- 4.1 What Does Step 1 Involve? 11
- 4.2 What Design Decisions Are Made During Step 1? 12

5 Step 2: Choose an Element of the System to Decompose 13
- 5.1 What Does Step 2 Involve? 13
- 5.2 What Design Decisions Are Made During Step 2? 14

6 Step 3: Identify Candidate Architectural Drivers 15
- 6.1 What Does Step 3 Involve? 15
- 6.2 What Design Decisions Are Made During Step 3? 15

7 Step 4: Choose a Design Concept That Satisfies the Architectural Drivers 17
- 7.1 What Does Step 4 Involve? 17
- 7.2 What Design Decisions Are Made During Step 4? 19

8 Step 5: Instantiate Architectural Elements and Allocate Responsibilities 21
- 8.1 What Does Step 5 Involve? 21
- 8.2 What Design Decisions Are Made During Step 5? 22

9 Step 6: Define Interfaces for Instantiated Elements 25
- 9.1 What Does Step 6 Involve? 25
- 9.2 What Design Decisions Are Made During Step 6? 25

10 Step 7: Verify and Refine Requirements and Make Them Constraints for Instantiated Elements 27
- 10.1 What Does Step 7 Involve? 27
- 10.2 What Design Decisions Are Made During Step 7? 27

11 Step 8: Repeat Steps 2 through 7 for the Next Element of the System You Wish to Decompose 29

12 Summary 31

Appendix A: ADD Checklist 33

Glossary 37

References 41

List of Figures

Figure 1: The ADD Plan, Do, and Check Cycle 4

Figure 2: Steps of ADD 5

List of Tables

Table 1: Structure of Matrix to Evaluate Candidate Patterns 18

Abstract

This report revises the Attribute-Driven Design (ADD) method that was developed by the Carnegie Mellon Software Engineering Institute. The motivation for revising ADD came from practitioners who use the method and want ADD to be easier to learn, understand, and apply.

The ADD method is an approach to defining a software architecture in which the design process is based on the software quality attribute requirements. ADD follows a recursive process that decomposes a system or system element by applying architectural tactics and patterns that satisfy its driving quality attribute requirements.

This technical report revises the steps of ADD and offers practical guidelines for carrying out each step. In addition, important design decisions that should be considered at each step are provided.

Acknowledgements

We express our thanks to the people who contributed to this revision of Attribute-Drive Design (ADD) by providing feedback on using the method, by reviewing drafts of the report, offering supporting materials, and coordinating and attending numerous and lengthy meetings, and through email, backroom, and hallway discussions. These people include Joe Batman, Linda Northrop, Mark Klein, Rick Kazman, Larry Jones, John Bergey, Ipek Ozkaya, Matt Bass, Carolyn Kernan, Pat McDonald, Laura Huber, Pennie Walters, and Alison Huml.

1 Introduction

This report revises the Attribute-Driven Design (ADD) method that was developed by the Carnegie Mellon® Software Engineering Institute (SEI). The motivation for refining ADD came from practitioners who use the method and want ADD to be easier to learn, understand, and apply. To these ends, the method has been revised by

- clarifying the inputs to and expected outputs from ADD
- renumbering and renaming the steps
- clarifying the purpose of each step
- offering guidelines on how to carry out each step
- identifying the design decisions that should be made at each step

This document is organized as follows:

- Section 2 provides a brief overview of ADD.
- Section 3 describes the inputs to and expected outputs from ADD.
- Sections 4 through 11 describe each step of ADD along with guidelines for carrying it out and the design decisions involved.
- Section 12 summarizes the findings of this report.
- Appendix A: ADD Checklist is an abbreviated checklist of the eight steps in the ADD method for your convenience.
- The Glossary section provides a glossary of the terms used in this report.
- The References section lists the references cited in this report.

® Carnegie Mellon is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

2 ADD Overview

The ADD method is an approach to defining a software architecture in which the design process is based on the software's quality attribute requirements. ADD follows a recursive design process that decomposes a system or system element by applying architectural tactics [Bass 03] and patterns that satisfy its driving requirements. As illustrated in Figure 1, ADD essentially follows a "Plan, Do, and Check" cycle:

- **Plan:** Quality attributes and design constraints are considered to select which types of elements will be used in the architecture.
- **Do:** Elements are instantiated to satisfy quality attribute requirements as well as functional requirements.
- **Check:** The resulting design is analyzed to determine if the requirements are met.

This process is repeated until all architecturally significant requirements are met.

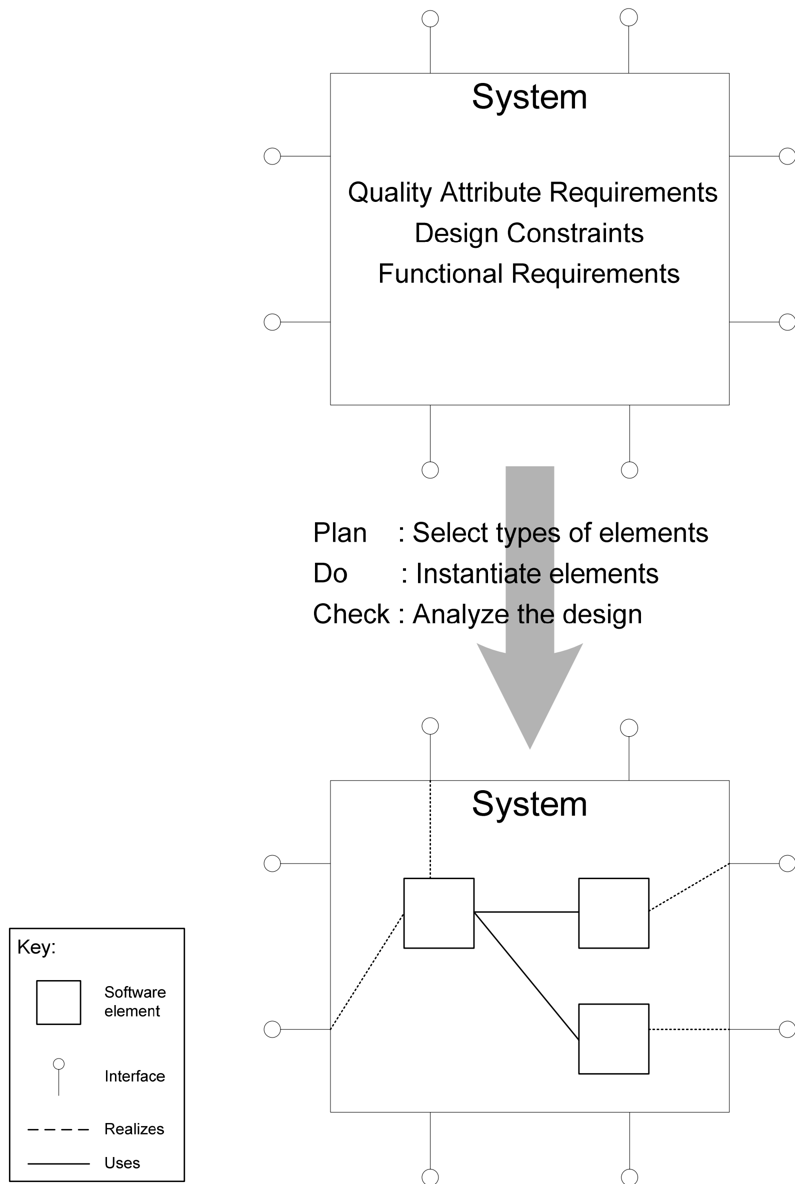


Figure 1: The ADD Plan, Do, and Check Cycle

Figure 2 provides an overview of the steps in ADD. Each step is described in detail in Sections 4 through 11 along with the design decisions made during each step.

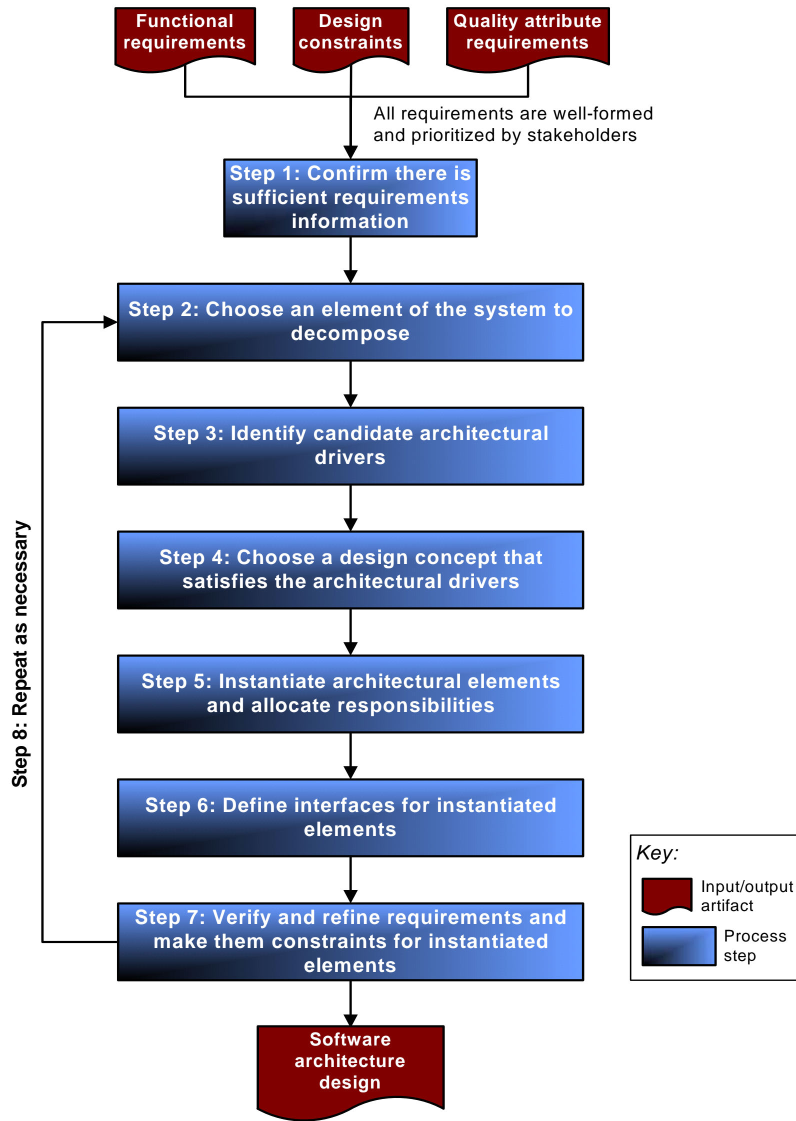


Figure 2: Steps of ADD

3 ADD Inputs and Outputs

3.1 INPUTS TO ADD

Inputs to ADD are functional requirements, design constraints, and quality attribute requirements that system stakeholders have prioritized according to business and mission goals.

Functional requirements specify what functions a system must provide to meet stated and implied stakeholder needs when the software is used under specific conditions [ISO 01]. For example, here is a sample list of functional requirements:[1]

- The system shall allow users to buy and sell securities.
- The system shall allow users to review account activity.
- The system shall monitor and record inputs from meteorological sensors.
- The system shall notify operators of reactor core temperature changes.
- The system shall compute and display the orbit and trajectory for all satellites.

Design constraints are decisions about a system's design that must be incorporated into any final design of the system. They represent a design decision with a predetermined outcome. For example, here is a sample list of design constraints:

- Oracle 8.0 shall be used for persistent storage.
- System services must be accessible through the World Wide Web.
- The system shall be implemented using Visual Basic.
- The system shall only interact with other systems via Publish/Subscribe.
- The system shall run on both Windows and Unix platforms.
- The system shall integrate with legacy applications.

Quality attribute requirements are requirements that indicate the degrees to which a system must exhibit various properties. For example, here is a sample list of quality attribute requirements:

- buildability: The system shall be buildable within six months.
- availability: The system shall recover from a processor crash within one second.
- portability: The system shall allow the user interface (UI) to be ported to a new platform within six months.

[1] The examples given for functional requirements, design constraints, and quality attribute requirements are meant to show the kind of requirements being discussed, and are not a recommendation of a specific requirements form to use with ADD. In practice, functional requirements are often captured as use cases. Quality attribute requirements can be captured as quality attribute scenarios [Bass 03].

- performance: The system shall process sensor input within one second.
- security: The system shall deny access to unauthorized users 100% of the time.
- testability: The system shall allow unit tests to be performed within three hours with 85% path coverage.
- usability: The system shall allow users to cancel an operation within one second.
- capacity: The system shall have a maximum of 50% CPU utilization.

Functional requirements, design constraints, and quality attribute requirements may be implied rather than explicit. For example, here are several implied constraints and requirements:

- "Given that all system transactions are subject to review by the Securities and Exchange Commission" implies that the system must keep a permanent transaction log and support report generation based on those transactions.
- "Given that Joe is the only resource available to handle the persistent storage aspects of the system and that Joe only has Oracle experience" implies that Oracle must be used for persistent storage.
- "Given that market demand for the system will increase dramatically within eight months" implies that our system must be buildable within six months.

In some cases, it may be difficult to categorize a particular condition as either a functional requirement, design constraint, or quality attribute requirement. For example, here are two uncategorized requirements:

- A security requirement to provide user authentication could be interpreted as a functional requirement or a quality attribute requirement.
- A requirement to interoperate with a particular legacy application could be interpreted as a functional requirement or a design constraint.

Fortunately, distinguishing among the three kinds of inputs is not as important as making sure that you collect these inputs before you begin ADD.

In the rest of this document, we refer to the above ADD inputs collectively as "the requirements." We also assume that we are addressing the system architect throughout the steps of the ADD method.

3.2 OUTPUTS TO EXPECT FROM ADD

The output of ADD is a system design in terms of the roles, responsibilities, properties, and relationships among software elements. The following terms are used throughout this document:

- **software element**: a computational or developmental artifact that fulfills various roles and responsibilities, has defined properties, and relates to other software elements to compose the architecture of a system [Bass 03]
- **role**: a set of related responsibilities [Wirfs-Brock 03]
- **responsibility**: the functionality, data, or information that a software element provides
- **property**: additional information about a software element such as name, type, quality attribute characteristic, protocol, and so on [Clements 03]
- **relationship**: a definition of how two software elements are associated with or interact with one another

The design that results from ADD is documented using various types of architectural views, including Module, Component-and-Connector, and Allocation views as appropriate [Clements 03]. In general, ADD produces an initial software architecture description from a set of design decisions to show

1. how to partition a system into major computational and developmental elements
2. what elements will be part of the different structures of the system, the type of each element, and the properties and structural relations they possess
3. what interactions will occur among elements, the properties of those interactions and the mechanisms by which they occur

Specific design decisions in each of the above categories are provided as we describe the steps of ADD in the rest of this document.

4 Step 1: Confirm There Is Sufficient Requirements Information

4.1 WHAT DOES STEP 1 INVOLVE?

In the first step, you confirm that there is sufficient information about the requirements to proceed with ADD.[2] In essence, you make sure that the system's stakeholders have prioritized the requirements according to business and mission goals. You should also confirm that there is sufficient information about the quality attribute requirements to proceed.

As the architect, you use the prioritized list of requirements to determine which system elements to focus on during the design.[3] You consider requirements and their potential impact on the architecture's structure in descending order of importance to stakeholders. Requirements that have not been prioritized should be flagged and returned to the stakeholders for ranking.

In addition, you should determine if there is sufficient information about the quality attribute requirements of the system. Each quality attribute requirement should be expressed in a "stimulus-response" form, in the same manner as quality attribute scenarios [Bass 03]. Each requirement should be described as the system's measurable quality attribute response to a specific stimulus with the following made explicit:

- stimulus source
- stimulus
- artifact
- environment
- response
- response measure

Knowing this information for each quality attribute requirement helps the architect to select various design patterns and tactics to achieve those requirements. If the above information is unavailable for a given quality attribute requirement, you should either create derived requirements or work with stakeholders to clarify the

---

[2] In many development scenarios, full and complete requirements are not known until quite late in the process. Requirements may be incomplete when ADD begins. ADD will produce an architectural design based on the requirements available at the time. The architect should mark design decisions based on requirements known to be tentative and be prepared to back up and revise the process based on better input.

[3] Stakeholders must prioritize the requirements prior to the first step of ADD; that prioritization process is outside the scope of ADD.

requirements. In any event, quality attribute scenarios can be used to document these requirements.

You can proceed with the design as long as the requirements have been prioritized by the stakeholders and there is sufficient information for one or more quality attribute requirements. The resulting design will be sufficient to the extent that the requirements gathered so far will influence the design of the architecture.

4.2 WHAT DESIGN DECISIONS ARE MADE DURING STEP 1?

No design decisions are made during this step.

5 Step 2: Choose an Element of the System to Decompose

5.1 WHAT DOES STEP 2 INVOLVE?

In this second step, you choose which element of the system will be the design focus in subsequent steps. You can arrive at this step in one of two ways:

1. You reach Step 2 for the first time as part of a "greenfield" development. The only element you can decompose is the system itself. By default, all requirements are assigned to that system.
2. You are refining a partially designed system and have visited Step 2 before.[4] In this case, the system has been partitioned into two or more elements, and requirements have been assigned to those elements. You must choose one of these elements as the focus of subsequent steps.

In the second case, you might choose the element to focus on based on one of these four areas of concern:

1. current knowledge of the architecture
   - if it is the only element you can choose (e.g., the entire system or the last element left)
   - the number of dependencies it has with other elements of the system (e.g., many or few dependencies)
2. risk and difficulty
   - how difficult it will be to achieve the element's associated requirements
   - how familiar you are with how to achieve the element's associated requirements
   - the risk involved with achieving the element's associated requirements
3. business criteria
   - the role the element plays in incremental development of the system
   - the role it plays in incremental releases of functionality (i.e., subsetability)
   - whether it will be built, purchased, licensed, or used as open source
   - the impact it has on time to market
   - whether it will be implemented using legacy components
   - the availability of personnel to address a component
4. organizational criteria
   - the impact it has on resource utilization (e.g., human and computing resources)

[4] The partial design may have come from previous iterations of ADD or from design constraints.

- the skill level involved with its development
- the impact it has on improving development skills in the organization
- someone of authority selected it

5.2 WHAT DESIGN DECISIONS ARE MADE DURING STEP 2?

No design decisions are made during this step.

6 Step 3: Identify Candidate Architectural Drivers

6.1 WHAT DOES STEP 3 INVOLVE?

At this point, you have chosen an element of the system to decompose, and stakeholders have prioritized any requirements that affect that element. During this step, you'll rank these same requirements a second time based on their relative impact on the architecture. This second ranking can be as simple as assigning "high impact," "medium impact," or "low impact" to each requirement.

Given that the stakeholders ranked the requirements initially, the second ranking based on architecture impact has the effect of partially ordering the requirements into a number of groups. If you use simple high/medium/low rankings, the groups would be

(H,H) (H,M) (H,L) (M,H) (M,M) (M,L) (L,H) (L,M) (L,L)

The first letter in each group indicates the importance of requirements to stakeholders. The second letter in each group indicates the potential impact of requirements on the architecture. Requirements in the (H,H) group are highly important to the stakeholders and are expected to have a high impact on the structure of the architecture, and so forth. From these pairs, you should choose several (five or six) high-priority requirements as the focus for subsequent steps in the design process.[5]

The selected requirements are called "candidate architectural drivers" for the element currently being decomposed. Further analysis may eliminate some candidates from consideration as architectural drivers while other requirements may be added to the candidate list. For example, although a requirement is ranked as having a high impact on the structure of the architecture, subsequent investigation may reveal that it does not. Analysis might also reveal that a requirement that was not considered to have a high impact on the architecture structure actually does, so it is added to the list of candidates. Ultimately, our goal in subsequent steps is to identify the true architectural drivers. The list of requirements that results from this step may or may not have an impact on the structure of the architecture. The ones that do will be called architectural drivers. Until then, they are called candidate architectural drivers.

6.2 WHAT DESIGN DECISIONS ARE MADE DURING STEP 3?

No design decisions are made during this step.

[5] If more than five or six requirements are ranked (H,H), this situation might signal a risk to the project and warrant renegotiation of the priorities. If renewed discussion is not feasible, then the architect should choose the handful of requirements that he/she believes will have the most far-reaching effect on the architecture.

7 Step 4: Choose a Design Concept That Satisfies the Architectural Drivers

7.1 WHAT DOES STEP 4 INVOLVE?

In Step 4, you should choose the major types of elements that will appear in the architecture and the types of relationships among them. Design constraints and quality attribute requirements (which are candidate architectural drivers) are used to determine the types of elements, relationships, and their interactions.

As the architect, you should follow these six sub-steps:

1. Identify the design concerns that are associated with the candidate architectural drivers. For example, for a quality attribute requirement regarding availability, the major design concerns might be fault prevention, fault detection, and fault recovery [Bass 03].
2. For each design concern, create a list of alternative patterns that address the concern.[6] Patterns on the list are derived from
   - your knowledge, skills, and experience about which patterns might be appropriate
   - known architectural tactics for achieving quality attributes [Bass 03]
     If a candidate architectural driver concerns more than one quality attribute, multiple tactics may apply.
   - other sources such as books, papers, conference materials, search engines, commercial products, and so forth

   For each pattern on your list, you should

   a. identify each pattern's discriminating parameters to help you choose among the patterns and tactics in the list.
      For example, in any restart pattern (e.g., warm restart, cold restart), the amount of time it takes for a restart is a discriminating parameter. For patterns used to achieve modifiability (e.g., layering), a discriminating parameter is the number of dependencies that exist between elements in the pattern.
   b. estimate the values of the discriminating parameters
3. Select patterns from the list that you feel are most appropriate for satisfying the candidate architectural drivers. Record the rationale for your selections.[7] To decide which patterns are appropriate

[6] Pattern purists may insist that a pattern is something that has been observed "in the wild" at least three times. We use the term *pattern* more loosely and include new inventions, too.

a. Create a matrix (as illustrated in Table 1) with patterns across the top and the candidate architectural drivers listed on the left-hand side. Use the matrix to analyze the advantages/disadvantages of applying each pattern to each candidate architectural driver. Consider the following:
   - What tradeoffs are expected when using each pattern?
   - How well do the patterns combine with each other?
   - Are any patterns mutually exclusive (i.e., you can use either pattern A or pattern B but not both)?

b. Choose patterns that together come closest to satisfying the architectural drivers.

Table 1: Structure of Matrix to Evaluate Candidate Patterns

	Pattern 1		**Pattern 2**		**. . .**	**Pattern n**	
	Pros	Cons	Pros	Cons		Pros	Cons
Architectural driver 1							
Architectural driver 2							
. . .							
Architectural driver n							

4. Consider the patterns identified so far and decide how they relate to each other. The combination of the selected patterns results in a new pattern.
   a. Decide how the types of elements from the various patterns are related.[8]
   b. Decide which types of elements from the various patterns are not related.
   c. Look for overlapping functionality and use it as an indicator for how to combine patterns.
   d. Identify new element types that emerge as a result of combining patterns.
   e. Review the list of design decisions at the end of this section and confirm that you have made all the relevant decisions.
5. Describe the patterns you've selected by starting to capture different architectural views, such as Module, Component-and-Connector, and Allocation views. You don't need to create fully documented architectural views at this

---

[7] Ideally, you want to find a pattern that satisfies all of your candidate drivers; otherwise, pick a pattern that comes close and augment it with tactics. If you can't find a pattern that comes close, start with tactics. We're using the term *pattern* to describe element types and their relationships.

[8] The relationships can be both runtime and non-runtime related. An example of a runtime relationship is to insert a "sends data to" relationship between the last filter in a Pipe-and-Filter pattern and a server in a Client-Server pattern. An example of a non-runtime (in this case, "is part of") relationship is to assign clients and servers to tiers in a Multi-Tier pattern.

point. Document any information that you are confident in or that you need to have to reason about the architecture (including what you know about the properties of the various element types). Ideally, you should use view templates to capture this information [Clements 03].

6. Evaluate and resolve inconsistencies in the design concept:
   a. Evaluate the design against the architectural drivers. If necessary, use models, experiments, simulations, formal analysis, and architecture evaluation methods.
   b. Determine if there are any architectural drivers that were not considered.
   c. Evaluate alternative patterns or apply additional tactics, if the design does not satisfy the architectural drivers.
   d. Evaluate the design of the current element against the design of other elements in the architecture and resolve any inconsistencies. For example, while designing the current element, you may discover certain properties that must be propagated to other elements in the architecture.

7.2 WHAT DESIGN DECISIONS ARE MADE DURING STEP 4?

The design concept you adopted in Step 4 will lead you to make (or at least consider) the following design decisions:[9]

- You have decided on an overall design concept that includes the major types of elements that will appear in the architecture and the types of relationships among them.
- You have identified some of the functionality associated with the different types of elements (e.g., elements that ping in a Ping-Echo pattern will have ping functionality).
- You know how and when particular types of software elements map to one another (i.e., either statically or dynamically).
- You have thought out communication among the various types of elements (both internal software elements and external entities) but perhaps deferred decisions about it. You have considered
  - which types of elements need to communicate with each other
  - what classes of mechanisms and protocols will be used for communication between software elements and external entities (e.g., synchronous, asynchronous, hybrid coupling, or remote versus local calls)
  - the required properties of mechanisms that will be used for communication between software elements and external entities (e.g., synchronous,

[9] All of these decisions may not be resolved by the design concept we adopt in this step. Some decisions may be deferred to another step, while others may be irrelevant to the particular system being developed.

asynchronous, hybrid coupling, throughput, queue capacity, and reliability)

  - the quality attribute requirements associated with the communication mechanisms
  - the data models on which communication depends
  - the types of computational elements support the various categories of system use
  - how legacy components and components off the shelf (COTS) will be integrated into the design

- You have reasoned about software elements and system resources but perhaps deferred decisions about them. You have considered
  - what resources are required by software elements
  - what resources need to be managed
  - the resource limits
  - how resources will be managed
  - what scheduling strategies will be employed
  - what elements are stateful/stateless
  - the major modes of operation
- You have thought out dependencies between the various types of internal software elements but perhaps deferred decisions about them. You have considered
  - what execution dependencies exist among elements
  - how and where execution dependencies among elements are resolved
  - the activation and deactivation dependencies among software elements
- You have also considered—and perhaps deferred decisions about—the following:
  - the abstraction mechanisms used
  - what system elements know about time
  - what process/thread model(s) will be employed
  - how quality attribute requirements will be addressed

8 Step 5: Instantiate Architectural Elements and Allocate Responsibilities

8.1 WHAT DOES STEP 5 INVOLVE?

In Step 5, you instantiate the various types of software elements you chose in the previous step. Instantiated elements are assigned responsibilities according to their types; for example, in a Ping-Echo pattern, a ping-type element has ping responsibilities and an echo-type element has echo responsibilities. Responsibilities for instantiated elements are also derived from the functional requirements associated with candidate architectural drivers and the functional requirements associated with the parent element. At the end of Step 5, every functional requirement associated with the parent element must be represented by a sequence of responsibilities within the child elements.

You should proceed with the following six sub-steps.

1. Instantiate one instance of every type of element you chose in Step 4. These instances are referred to as "children" or "child elements" of the element you are currently decomposing (i.e., the parent element).
2. Assign responsibilities to child elements according to their type. For example, ping-type elements are assigned responsibilities including ping functionality, ping frequency, data content of ping signals, and the elements to which they send ping signals.
3. Allocate responsibilities associated with the parent element among its children according to the rationale and element properties recorded in Step 4. For example, if a parent element in a banking system is responsible for managing cash distribution, cash collection, and transaction records, then allocate those responsibilities among its children. Note that all responsibilities assigned to the parent are considered at this time regardless of whether they are architecturally significant.

   At this point, it may be useful to consider use cases that systems typically address—regardless of whether they were given explicitly as requirements. This exercise might reveal new responsibilities (e.g., resource management). In addition, you might discover new element types and wish to create new instances of them. These use cases include

   - one user doing two tasks simultaneously
   - two users doing similar tasks simultaneously
   - startup
   - shutdown
   - disconnected operation
   - failure of various elements (e.g., the network, processor, process)

– version upgrades

4. Create additional instances of element types in these two circumstances:
   a. A difference exists in the quality attribute properties of the responsibilities assigned to an element. For example, if a child element is responsible for collecting sensor data in real time and transmitting a summary of that data at a later time, performance requirements associated with data collection may prompt us to instantiate a new element to handle data collection while the original element handles transmitting a summary.
   b. You want to achieve other quality attribute requirements; for example, you reassign the functionality of one element to two elements to promote modifiability.

   At this point, you should review the list of design decisions listed at the end of this section and confirm that you made all the relevant decisions.

5. Analyze and document the design decisions you have made during step 5 using various views such as these three:
   a. Module views are useful for reasoning about and documenting the non-runtime properties of a system (e.g., modifiability).
   b. Component-and-Connector views are useful for reasoning about and documenting the runtime behaviors and properties of a system (e.g., how elements will interact with each other at runtime to meet various requirements and what performance characteristics those elements should exhibit).
   c. Allocation views are useful for reasoning about the relationships between software and non-software (e.g., how software elements will be allocated to hardware elements).

8.2 WHAT DESIGN DECISIONS ARE MADE DURING STEP 5?

Some of the decisions may not be relevant to your particular type of system. However, your decisions will likely involve several of the following:

- how many of each type of element will be instantiated and what individual properties and structural relations they will possess
- what computational elements will be used to support the various categories of system use
- what elements will support the major modes of operation
- how quality attribute requirements have been satisfied within the infrastructure and applications
- how functionality is divided and assigned to software elements including how functionality is allocated across the infrastructure and applications

- how software elements map to each other
  - how system elements in different architectural structures map to each other (e.g., how modules map to runtime elements and how runtime elements map to processors)
  - whether the mapping of one system element to another is static or dynamic (i.e., when the mapping is determined—at build time, deployment, load time, or runtime)
- communication among the various elements, both internal software elements and external entities
  - which software elements need to communicate with each other
  - what mechanisms and protocols will be used for communication between software elements and external entities
  - the required properties of mechanisms that will be used for communication between software elements and external entities (e.g., synchronous, asynchronous, hybrid coupling)
  - the quality attribute requirements associated with the communication mechanisms
  - the data models on which communication depends
  - what computational elements support the various categories of system use
  - how legacy and COTS components will be integrated into the design
- internal software elements and system resources
  - what resources are required by software elements
  - what resources need to be managed
  - the resource limits
  - how resources will be managed
  - what scheduling strategies will be employed
  - what elements are stateful/stateless
  - the major modes of operation
- dependencies between the internal software elements
  - what execution dependencies exist among elements
  - how and where execution dependencies among elements are resolved
  - the activation and deactivation dependencies among software elements
- which abstraction mechanisms are used
- how much system elements know about time
- what process/thread model(s) will be employed
- how quality attribute requirements will be addressed

9 Step 6: Define Interfaces for Instantiated Elements

9.1 WHAT DOES STEP 6 INVOLVE?

In step 6, you define the services and properties required and provided by the software elements in our design. In ADD, these services and properties are referred to as the element's interface. Note that an interface is not simply a list of operation signatures. Interfaces describe the PROVIDES and REQUIRES assumptions that software elements make about one another. An interface might include any of the following:

- syntax of operations (e.g., signature)
- semantics of operations (e.g., description, pre- and postconditions, restrictions)
- information exchanged (e.g., events signaled, global data)
- quality attribute requirements of individual elements or operations
- error handling

You should proceed with these three steps:

1. Exercise the functional requirements that involve the elements you instantiated in Step 5.
2. Observe any information that is produced by one element and consumed by another. Consider the interfaces from the perspective of different views. For example, a Module view will allow you to reason about information flow; a Concurrency view will allow you to reason about performance and availability; and a Deployment view will allow you to reason about security and availability.
3. Record your findings in the interface documentation for each element.

9.2 WHAT DESIGN DECISIONS ARE MADE DURING STEP 6?

Some of the decisions may not be relevant to your particular type of system. However, your decisions will likely involve several of the following:

- the external interfaces to the system
- the interfaces between high-level system partitions
- the interfaces between applications within high-level system partitions
- the interfaces to the infrastructure

10 Step 7: Verify and Refine Requirements and Make Them Constraints for Instantiated Elements

10.1 WHAT DOES STEP 7 INVOLVE?

In Step 7, you verify that the element decomposition thus far meets functional requirements, quality attribute requirements, and design constraints. You also prepare child elements for further decomposition.

You should proceed with these three sub-steps:

1. Verify that all functional requirements, quality attribute requirements, and design constraints assigned to the parent element have been allocated to one or more child elements in the decomposition.
2. Translate any responsibilities that were assigned to child elements into functional requirements for the individual elements.
3. Refine quality attribute requirements for individual child elements as necessary.

10.2 WHAT DESIGN DECISIONS ARE MADE DURING STEP 7?

No design decisions are made during this step.

11 Step 8: Repeat Steps 2 through 7 for the Next Element of the System You Wish to Decompose

Once you have completed Steps 1–7, you have a decomposition of the parent element into child elements. Each child element is a collection of responsibilities, each having an interface description, functional requirements, quality attribute requirements, and design constraints. You can now return to the decomposition process in Step 2 where you select the next element to decompose.

12 Summary

This report revises the ADD method to make the method easier for practitioners to learn, understand, and apply. While the method has been used successfully prior to these revisions, we hope that our work here will offer a deeper understanding to those who have already applied it and make the method more accessible to those who have not.

ADD is a powerful method for architecture design that distinguishes itself from other design methods because it focuses on system decomposition from a quality attributes' perspective. ADD can be used in conjunction with other SEI methods such as the Quality Attribute Workshop for gathering requirements for input to ADD or the Architecture Tradeoff Analysis Method® (ATAM®) to evaluate architectures that result from applying ADD. ADD can also be used within or adapted to most any development life cycle or process (e.g., evolutionary, waterfall, spiral, Rational Unified Process [RUP], or agile development).

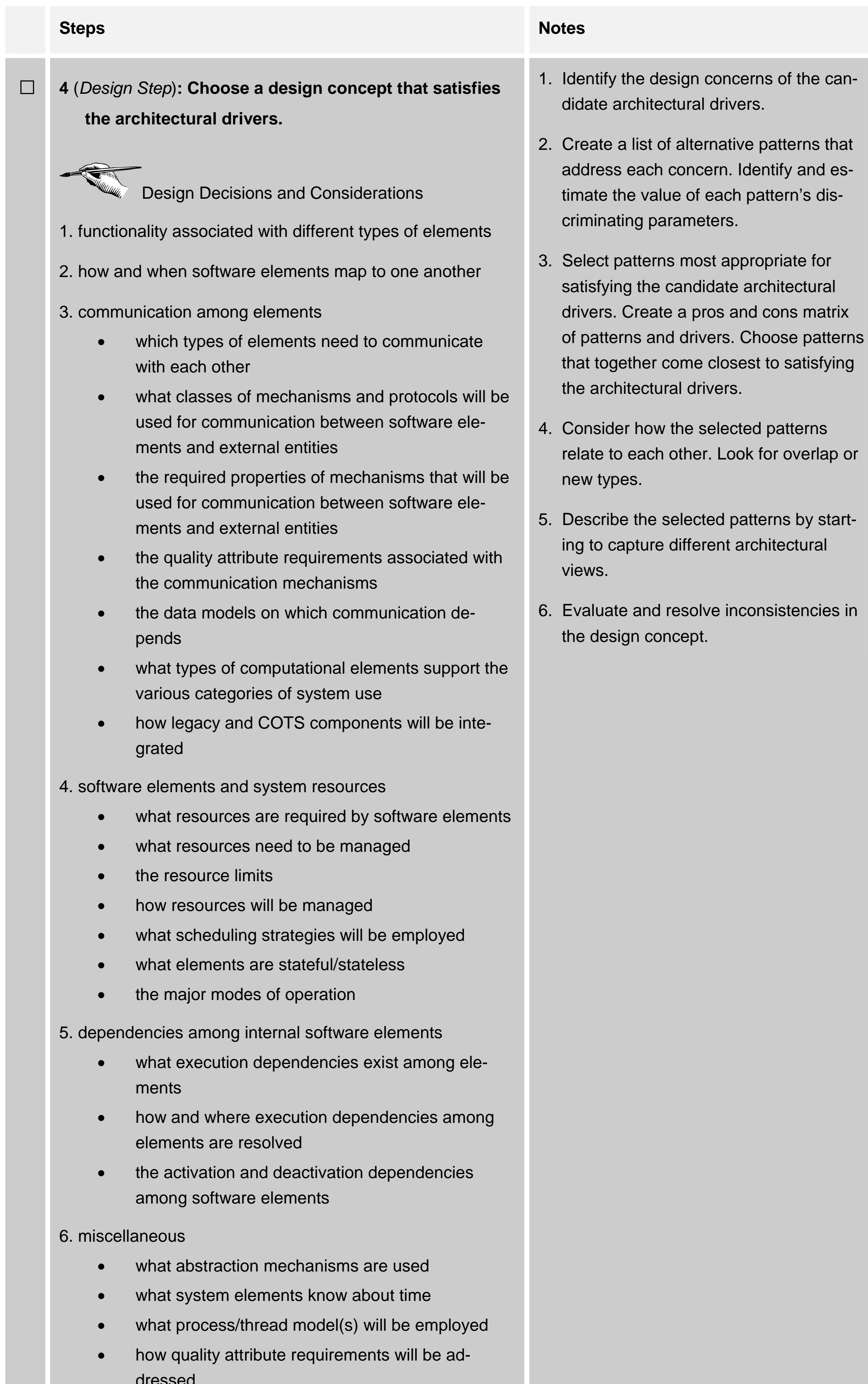
Although we consider this report to be a major revision to ADD, our intent is to continue watching over and improving the method as we learn of its strengths and weaknesses through our own applications as well as through feedback we receive from other practitioners. We are grateful to those who have told us about their experiences using the method, and we welcome comments and suggestions on this document and the use of the method to aid our next revision.

® Architecture Tradeoff Analysis Method and ATAM are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

Appendix A: ADD Checklist

For your convenience, here is an abbreviated checklist of the eight steps in the ADD method.

	Steps	Notes
☐	**0: System's stakeholders prioritize the requirements according to business and mission goals.**	*Outside the scope of ADD*
☐	**1: Confirm that there is sufficient requirements information.**	Use the prioritized requirements list to determine which system elements to focus on during design. Consider elements and their impact on the architecture's structure in descending order of importance to stakeholders. Return unranked requirements to the stakeholders for prioritization. Each quality attribute requirement should be expressed in a "stimulus-response" form.
☐	**2: Choose an element of the system to decompose.**	If this is your first iteration of Step 2, decompose the entire system. If not, you select the element to decompose based on 1. current knowledge of the architecture 2. risk and difficulty 3. business criteria 4. organizational criteria
☐	**3: Identify candidate architectural drivers.**	1. Rank requirements a second time according to their impact on the architecture. 2. Group your priorities based on both their importance to stakeholders and their architecture impact; that is, if you use simple high (H), medium (M), and low (L) rankings, you have 9 groups: (H,H) (H,M) (H,L) (M,H) (M,M) (M,L) (L,H) (L,M) (L,L) 3. Select five or six high-priority requirements as the candidate architectural drivers.



	Steps	Notes
☐	**4** (*Design Step*)**: Choose a design concept that satisfies the architectural drivers.** Design Decisions and Considerations 1. functionality associated with different types of elements 2. how and when software elements map to one another 3. communication among elements • which types of elements need to communicate with each other • what classes of mechanisms and protocols will be used for communication between software elements and external entities • the required properties of mechanisms that will be used for communication between software elements and external entities • the quality attribute requirements associated with the communication mechanisms • the data models on which communication depends • what types of computational elements support the various categories of system use • how legacy and COTS components will be integrated 4. software elements and system resources • what resources are required by software elements • what resources need to be managed • the resource limits • how resources will be managed • what scheduling strategies will be employed • what elements are stateful/stateless • the major modes of operation 5. dependencies among internal software elements • what execution dependencies exist among elements • how and where execution dependencies among elements are resolved • the activation and deactivation dependencies among software elements 6. miscellaneous • what abstraction mechanisms are used • what system elements know about time • what process/thread model(s) will be employed • how quality attribute requirements will be addressed	1. Identify the design concerns of the candidate architectural drivers. 2. Create a list of alternative patterns that address each concern. Identify and estimate the value of each pattern's discriminating parameters. 3. Select patterns most appropriate for satisfying the candidate architectural drivers. Create a pros and cons matrix of patterns and drivers. Choose patterns that together come closest to satisfying the architectural drivers. 4. Consider how the selected patterns relate to each other. Look for overlap or new types. 5. Describe the selected patterns by starting to capture different architectural views. 6. Evaluate and resolve inconsistencies in the design concept.

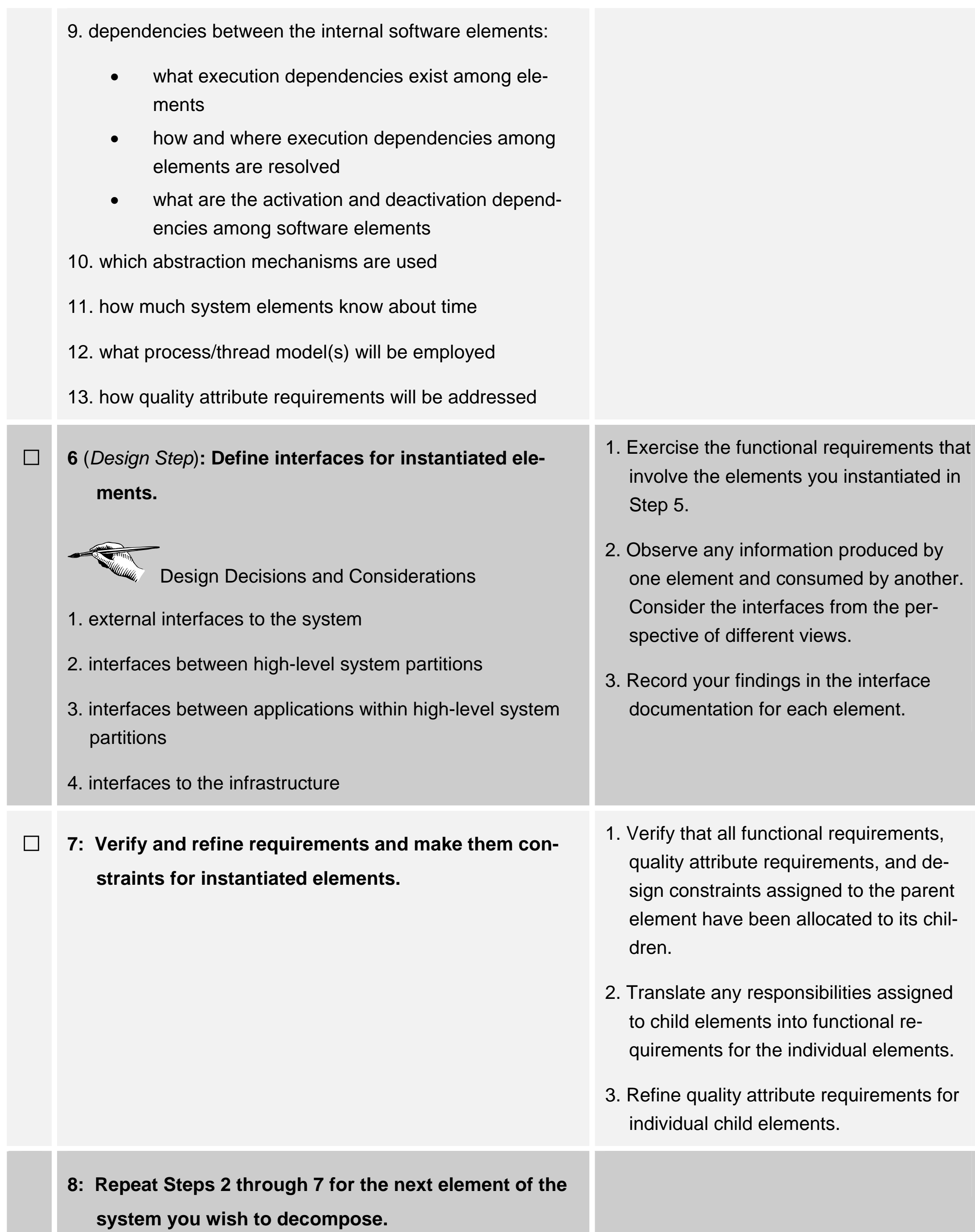
☐ **5** (*Design Step*): **Instantiate architectural elements and allocate responsibilities.**

Design Decisions and Considerations

1. how many of each type of element will be instantiated, and what properties and structural relations they possess
2. what computational elements will be used to support the various categories of system use
3. what elements will support the major modes of operation
4. how quality attribute requirements have been satisfied within the infrastructure and applications
5. how functionality is divided and assigned to software elements including how functionality is allocated across the infrastructure and applications
6. how software elements map to each other:
   - how system elements in different architectural structures map to each other
   - whether the mapping of one system element to another is static or dynamic
7. communication among the various elements, both internal software elements and external entities
   - which software elements need to communicate with each other
   - what mechanisms and protocols will be used for communication between software elements and external entities
   - the required properties of mechanisms that will be used for communication between software elements and external entities
   - the quality attribute requirements associated with the communication mechanisms
   - the data models on which communication depends
   - what computational elements support the various categories of system use
   - how legacy and COTS components will be integrated into the design
8. internal software elements and system resources
   - what resources are required by software elements
   - what resources need to be managed
   - the resource limits
   - how resources will be managed
   - what scheduling strategies will be employed
   - what elements are stateful/stateless
   - the major modes of operation

1. Instantiate one instance (“child”) of every type of element you selected in Step 4.
2. Assign responsibilities to child elements according to their type.
3. Allocate the parent’s responsibilities among its children according to the rationale and element properties (recorded in Step 4). Consider use cases:
   - one user doing two tasks simultaneously
   - two users doing similar tasks simultaneously
   - startup
   - shutdown
   - disconnected operation
   - failure of various elements
   - version upgrades
4. Create additional instances of element types if (a) a difference exists in the quality attribute properties of the responsibilities assigned to an element, or (b) you want to achieve other quality attribute requirements.

   Review the design decisions listed to the left and confirm that you made all the relevant decisions.
5. Analyze and document the design decisions using the three views:
   - Module
   - Component-and-Connector
   - Allocation



	9. dependencies between the internal software elements: • what execution dependencies exist among elements • how and where execution dependencies among elements are resolved • what are the activation and deactivation dependencies among software elements 10. which abstraction mechanisms are used 11. how much system elements know about time 12. what process/thread model(s) will be employed 13. how quality attribute requirements will be addressed	
☐	**6** (*Design Step*)**: Define interfaces for instantiated elements.** Design Decisions and Considerations 1. external interfaces to the system 2. interfaces between high-level system partitions 3. interfaces between applications within high-level system partitions 4. interfaces to the infrastructure	1. Exercise the functional requirements that involve the elements you instantiated in Step 5. 2. Observe any information produced by one element and consumed by another. Consider the interfaces from the perspective of different views. 3. Record your findings in the interface documentation for each element.
☐	**7: Verify and refine requirements and make them constraints for instantiated elements.**	1. Verify that all functional requirements, quality attribute requirements, and design constraints assigned to the parent element have been allocated to its children. 2. Translate any responsibilities assigned to child elements into functional requirements for the individual elements. 3. Refine quality attribute requirements for individual child elements.
	8: Repeat Steps 2 through 7 for the next element of the system you wish to decompose.	

Glossary

architectural driver
An architectural driver is any functional requirement, design constraint, or quality attribute requirement that has a significant impact on the structure of an architecture.

architectural patterns
Architectural patterns are well-known ways to solve recurring design problems. For example, the Layered and Model-View-Controller patterns help to address design problems related to modifiability. Patterns are typically described in terms of their elements, the relationships between elements, and usage rules.

architectural tactics
Architectural tactics are design decisions that influence the quality attribute properties of a system. For example, a Ping-Echo tactic for fault detection may be employed during design to influence the availability properties of a system. The Hide Information tactic may be employed during design to influence the modifiability properties of a system.

candidate architectural driver
Candidate architectural drivers are any functional requirements, design constraints, or quality attribute requirements that have a potentially significant impact on the structure of an architecture. Further analysis of such requirements during design may reveal that they have no significant impact on the architecture.

design concept
A design concept is an overview of an architecture that describes the major types of elements that appear in the architecture and the types of relationships between them.

design concern
Design concerns are specific problem areas that must be addressed during design. For example, for a quality attribute requirement regarding availability, the major design concerns are fault prevention, fault detection, and fault recovery. For a quality attribute requirement regarding availability, the major design concerns are resource demand, resource management, and resource arbitration [Bass 03].

design constraints
Design constraints are decisions about the design of a system that must be incorporated into any final design of the system. They represent a design decision with a predetermined outcome.

discriminating parameter

Discriminating parameters are characteristics of patterns that you evaluate to determine if those patterns help you achieve the quality attribute requirements of a system. For example, in any restart pattern (e.g., Warm Restart, Cold Restart), the amount of time to takes to do a restart is a discriminating parameter. For patterns used to achieve modifiability (e.g., Layered), a discriminating parameter is the number of dependencies that exist between elements in the pattern.

functional requirements

Functional requirements specify what functions a system must provide to meet stated and implied stakeholders' needs when the software is used under specific conditions [ISO 01].

interface

The interface for an element refers to the services and properties required and provided by that element. Note that *interface* is not synonymous with *signature*. Interfaces describe the *provides* and *requires* assumptions that software elements make about one another. An interface specification for an element is a statement of an element's properties that the architect chooses to make known [Bass 03].

patterns

See **architectural patterns**.

property

A property is additional information about a software element such as name, type, quality attribute characteristic, protocol, and so forth. [Clements 03].

quality attribute

A quality attribute is a property of a work product or goods by which its quality will be judged by stakeholders. Quality attribute requirements such as those for performance, security, modifiability, reliability, and usability have a significant influence on the software architecture of a system [SEI 06].

quality attribute requirements

Quality attribute requirements are requirements that indicate the degrees to which a system must exhibit various properties.

relationship

A relationship defines how two software elements are associated with or interact with one another.

requirements

Requirements are the functional requirements, design constraints, and quality attribute requirements that a system must satisfy for a software system to meet mission/business goals and objectives.

responsibility

A responsibility is functionality, data, or information that is provided by a software element.

role

A role is a set of related responsibilities [Wirfs-Brock 03].

software architecture

The software architecture of a program or computing system is the structure(s) of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them [Bass 03].

software element

A software element is a computational or developmental artifact that fulfills various roles and responsibilities, has defined properties, and relates to other software elements to compose the architecture of a system.

stakeholder

A stakeholder is someone who has a vested interest in an architecture [SEI 06].

tactics

See **architectural tactics**.

References

URLs are valid as of the publication date of this document.

[Bass 03]
Bass, L.; Clements, P.; & Kazman, R. *Software Architecture in Practice*. Reading, MA: Addison-Wesley, 2003.

[Clements 03]
Clements, P.; et al. *Documenting Software Architectures Views and Beyond*. Reading, MA: Addison-Wesley, 2003.

[ISO 01]
International Organization for Standardization *Software engineering–Product quality–Part 1: Quality model*. ISO/IEC 9126-1:2001
http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=22749 (June 15, 2001).

[SEI 06]
Software Engineering Institute. *Software Architecture Glossary*.
http://www.sei.cmu.edu/architecture/glossary.html (2006).

[Wirfs-Brock 03]
Wirfs-Brock, Rebecca; McKean, Alan *Object Design Roles, Responsibilities, and Collaborations*. Boston, MA: Addison-Wesley, 2003.

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	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	November 2006	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Attribute-Driven Design (ADD), Version 2.0	FA8721-05-C-0003

6. AUTHOR(S)

Rob Wojcik, Felix Bachmann, Len Bass, Paul Clements, Paulo Merson, Robert Nord, & Bill Wood

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2006-TR-023

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116	ESC-TR-2006-023

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

This report revises the Attribute-Driven Design (ADD) method that was developed by the Carnegie Mellon Software Engineering Institute. The motivation for revising ADD came from practitioners who use the method and want ADD to be easier to learn, understand, and apply.

The ADD method is an approach to defining a software architecture in which the design process is based on the software quality attribute requirements. ADD follows a recursive process that decomposes a system or system element by applying architectural tactics and patterns that satisfy its driving quality attribute requirements.

This technical report revises the steps of ADD and offers practical guidelines for carrying out each step. In addition, important design decisions that should be considered at each step are provided.

14. SUBJECT TERMS	15. NUMBER OF PAGES
attribute-driven design, ADD, architectural drivers, software architecture, architecturally significant requirements, decomposition	55

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL