Technical Report
CMU/SEI-88-TR-008
ESD-TR-88-009

A Guide to the Assessment of Software Development Methods

Bill Wood
Richard Pethia
Lauren Roberts Gold
Robert Firth
April 1988

Preliminary Report

CMU/SEI-88-TR-008
ESD-TR-88-009
April 1988

A Guide to the Assessment of Software Development Methods



Bill Wood
Richard Pethia
Lauren Roberts Gold
Robert Firth

Tools and Methodologies for Real-Time Systems Project

Unlimited distribution subject to the copyright.

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.

Copyright 1988 by Carnegie Mellon University.

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

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 MERCHANTIBILITY, 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.

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 Asset Source for Software Engineering Technology (ASSET) / 1350 Earl L. Core Road ; P.O. Box 3305 / Morgantown, West Virginia 26505 / Phone: (304) 284-9000 / Fax: (304) 284-9001 / e-mail: sei@asset.com / WWW: http://www.asset.com/sei.html

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 acess to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential con tractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center / 8725 John J. Kingman Road / Suite 0944 / Ft. Belvoir, VA 22060-6218. Phone: 1-800-225-3842 or 703-767-8222.

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

Table of Contents

1. Introduction **1**

2. Context **3**
2.1. Key Aspects 3
2.2. Development Stages 3
2.3. Views of the System 4
2.4. Classification Scheme 5

3. System Characteristics **7**
3.1. Operational Characteristics 8
3.1.1. Functional 8
3.1.1.1. Environment 8
3.1.1.2. I/O 8
3.1.1.3. Data Transformations 9
3.1.1.4. Math Representations of Engineering Phenomena 9
3.1.2. Behavioral 9
3.1.2.1. Modes and States 9
3.1.2.2. Capacity, Workload, and Performance 10
3.1.2.3. Human Interface 11
3.2. Structural Characteristics 12
3.2.1. System Architecture 12
3.2.1.1. Distributed Processing 12
3.2.1.2. Robustness 12
3.2.2. Data Modeling 13
3.2.3. Language Platform 13

4. Constraints **15**
4.1. Software Architecture 15
4.1.1. Modularity 16
4.1.1.1. Size and Complexity 16
4.1.1.2. Coupling 16
4.1.1.3. Cohesion 17
4.1.2. Information Hiding 17
4.1.3. Exception Handling 17
4.2. Integration and Test Constraints 18
4.3. Evolution Constraints 18

5. Representations **21**
5.1. Abstraction 21
5.2. Consistency 22
5.3. Completeness 22
5.4. Complexity 22
5.5. Traceability 23
5.6. View Integration 23

5.7. Ambiguity 24
5.8. Duplication 24
5.9. Changes 24
5.10. Compliance with Standards 25

6. Deriving Representations 27
6.1. Partitioning 27
6.2. Refinement 28
6.3. Elaboration 28
6.4. Reuse 29
6.5. Evaluation of Alternatives 29

7. Examining Representations 31
7.1. Examination Goals 31
7.1.1. Feasibility 31
7.1.2. Conformance 32
7.1.3. Safety 32
7.2. Examination Techniques 32
7.2.1. Walkthroughs and Inspections 33
7.2.2. Analysis 33
7.2.3. Testing 34
7.2.4. Data Extraction, Reduction, and Analysis 34

8. Management Characteristics 35
8.1. Process 35
8.2. Cost 36

9. Problem Areas 39
9.1. Experienced Personnel 39
9.2. Transformation Across Stages 40
9.3. Feasibility Analysis 41
9.4. Development Constraints 42
9.5. Large-Scale Problems 43
9.6. User Interface 43
9.7. Implementing Designs 44

10. Conclusion 47

List of Tables

Table 2-1: Stages of Development 6

A Guide to the Assessment of Software Development Methods

Abstract. Over the past decade, the term "software engineering method" has been attached to a variety of procedures and techniques that attempt to provide an orderly, systematic way of developing software. Existing methods approach the task of software engineering in different ways. Deciding which methods to use to reduce development costs and improve the quality of produced products is a difficult task. This report outlines a five step process and an organized set of questions that provide method assessors with a systematic way to improve their understanding and form opinions on the ability of existing methods to meet their organization's needs.

1. Introduction

This is the third report in a series of reports concerned with classification and assessment criteria for software development methods and tools. The first two reports describe guidelines for classifying and evaluating software engineering tools [Firth 87a], and a classification scheme for software development methods [Firth 87b]. This (third) report describes issues in assessing methods for use in the specification and design of real-time software systems. Throughout this report, the term "development" is used in the restricted sense of specification and design.

Over the past decade, the term "software engineering method" has been attached to a variety of procedures and techniques that attempt to provide an orderly, systematic way of developing software. Existing methods such as Design Approach for Real-Time Systems [Gomaa 86]; Problem Statement Language/Problem Statement Analyzer [Teichrow 77]; Jackson System Design [Cameron 83]; Object-Oriented Design [Booch 83]; and Structured Analysis, Structured Development with Real-Time Extensions [Ward 85] (see [Firth 87b] for additional examples); approach the task of software engineering in different ways and prescribe different techniques. Some, such as Distributed Computing Design System [Alford 85], attempt to cover a broad range of activities while others, such as Statecharts [Harel 86], focus on particular areas that traditionally have been especially troublesome.

An assessor of methods is basically concerned with answering the question: What is a good software engineering method that my organization can use to reduce development costs and produce quality products? Answering this seemingly simple question is, however, a difficult task given the diversity of existing methods and the complexity of software engineering. In addition, methods alone provide no guarantees of productivity improvement or of high quality in the software product. Methods, procedures, and techniques are valuable only if those who use them understand their underlying concepts and apply them with thought and judgement.

The rest of this report describes a five step process (outlined in [Firth 87b]) to be used in evaluating methods. The five steps are:

1. **Needs Analysis** - Determine the important characteristics of the system to be developed and how individual methods help developers deal with those

characteristics. Chapter 3 emphasizes the need to evaluate individual methods by their ability to deal with specific engineering problems.

2. **Constraint Identification** - Identify the constraints imposed on the permitted solutions and determine how individual methods help developers deal with those constraints. Chapter 4 reminds the assessor that methods must support developers' efforts to design systems that exhibit a variety of required characteristics.

3. **User Requirements** - Determine the general usage characteristics of the individual methods. As described in Chapter 2, a method can be examined by developing an understanding of: how it represents a system under development, the guidelines it gives developers to derive the representations, and the guidelines it provides to examine the representations. This understanding is best developed by applying the method to a **sample problem** that is representative of the system to be developed. Chapters 5, 6, and 7 discuss representations, deriving representations and examining representations in turn.

4. **Management Issues** - Determine the support provided by the method to those who must manage the development process as well as the costs and benefits of adopting and using the method. Chapter 8 emphasizes the need to recognize that methods are used within particular organizations that have established ways of conducting business.

5. **Introduction Plan** - Develop an understanding of the issues that the method does not address and a plan to augment the method in areas where it is deficient. Chapter 9 reinforces the notion that methods do not guarantee success and points out several problem areas in existing methods and their use.

Chapters 3 through 9 of this report each contain an organized set of topics supported by a list of questions for each topic. The questions have the following characteristics.

1. Some are rhetorical and do not require answers. These serve to remind the assessor of important software engineering issues and the need to form opinions on how individual methods address the issues.

2. Some require an understanding of the method alone and emphasize the need to examine individual methods thoroughly.

3. Some require that the method be applied to a problem or set of problems; demonstrating the need to match the method to the engineering problem at hand.

4. Some require that the assessor form opinions on the needs of those involved in the development process and how they would use a method to satisfy those needs. These emphasize the need to use selected methods with judgement and understanding.

There has been no attempt to reduce the evaluation of methods to a completely objective process supported by a list of questions with quantifiable answers. To do so would oversimplify what is and must be viewed as a difficult task in which a large number of tradeoff decisions must be made. By using the framework and questions provided in this document, method assessors will have a systematic way to improve their understanding and form opinions on the ability of existing methods to meet their organization's needs.

2. Context

This chapter contains a discussion of a framework for the comparison of methods. The framework is introduced here to lend focus to the remainder of the report and to introduce terms that are used in later chapters. The key aspects of the methods are discussed first, then a two dimensional method classification scheme is introduced. A more detailed discussion of methods is available in [Firth 87b].

2.1. Key Aspects

In general, a method is "a systematic procedure, technique, or mode of inquiry employed by or proper to a particular discipline or art [Webster 81]." When applied specifically to software engineering, it could be defined as a systematic approach to providing a software solution. Ideally, the method should cover all aspects of the problem and, in context of software engineering, should lead from and initial (imperfect) set of requirements to a satisfactory implementation, passing systematically through intermediate stages.

At all stages in the development process, a representation of the system must be created. It is important to understand how the method contributes to this process. A detailed understanding of individual methods can be obtained by examining each method from three points of view, namely:

1. What is the **content and form of representation** of the artifacts dictated by the method? Desirable characteristics of representations are discussed in Chapter 5.
2. What procedures or techniques does the method provide for **deriving** the representations? Chapter 6 addresses the issue of deriving representations, and discusses five major areas where a method should provide guidance.
3. What procedures or techniques does the the method provide for **examining** the representations? Representations are typically examined in a variety of ways for several reasons. Chapter 7 discusses examinations and elaborates on the goals of examinations and the techniques for conducting examinations.

2.2. Development Stages

One axis of the classification scheme deals with the software development process and a large-grained view of development stages. The different development stages are enumerated below. Each development stage is characterized by what it represents and the way it represents it.

1. The **requirement** is a description of what the end-user audiences view as their needs and is often a rather eclectic description. It usually covers the needs of each audience in the end-user community in a very uneven manner, with some aspects (such as ease of installation) often overlooked. It often describes some functions (such as a scheduling mechanism) in very general terms, but others (such as a communications protocol) are discussed thoroughly. There are often important requirements that are not addressed, and they need to be exposed and handled as they arise. The earlier these issues are addressed and resolved, the better the prospect of delivering a high-quality product within budget and schedule.

2. The first step is to take the ambiguous, incomplete, and inconsistent requirement and turn it into an almost flawless **specification**. This is not yet possible with current technology, but there are many reasonable ways of proceeding that give a serviceable specification. The specification describes what the software is to do and the constraints to be imposed on the designers. It should be noted that production of the specification is not limited to a front-end activity, but that the specification will change throughout the life cycle of the system.

3. The **design** representation describes how the system is structured to satisfy the specification. It describes the system in a large-grained manner and defines the breakup of the system into major tasks. It describes persistent data objects and their access mechanisms, the important abstract data types and their encapsulation in the heavyweight tasks, and the message structures between the tasks. There must also be some consideration for how the resources are to be allocated and how the performance requirements are to be satisfied.

4. The final development stage is **implementation** with source code, object code, resource usage, and initialized data structures. This is the level at which algorithms are represented explicitly.

2.3. Views of the System

The second axis of the classification scheme deals with various views of the system. As suggested in [Harel 86] these views are needed to describe the system's intended and actual operation. The views are relevant at each stage of the system's development and are enumerated below.

1. The **functional** view shows the system as a set of processes operating on data. This includes a description of the task performed by each process, the flow of data between processes, and the underlying math model, if required. The functional view is often the starting point for the design process, since it deals with what the system is supposed to do, relates the system to its environment, and focuses on the needs of the key participants in the development process: customers, users, and developers.

2. The **structural** view shows how the system is put together: the system's components, the interfaces between them, and the distribution and flow of data and control between the components through the interfaces. It also shows the environment and the interfaces and information flows between it and the system. Ideally, the structural view should be an elaboration of the functional view. Each entity in the latter view is decomposed into a set of primitive software components that can be implemented separately and then combined to build the entity. The design process therefore generally converts a functional view into a structural view. However, the structure of a system is influenced by resource constraints that prevent the use of arbitrarily many or arbitrarily large components. The structure is also influenced by certain implementation constraints that require the use of specific types of component (e.g., MIL-STD-1750a processors), or require that components be connected in a specific manner (e.g., by MIL-STD-1553 buses). The structural view should include a definition of the number and dimensions of entities to allow for resource estimates.

3. The **behavioral** view shows the way the system will respond to specific inputs: what states it will adopt, what outputs it will produce for each combination and time sequence of inputs and state transitions, what boundary conditions exist on the validity of inputs and states. This includes a description of the environment that is

producing the inputs and consuming the outputs. It also includes constraints on performance that are imposed by the environment and function of the system. Real-time systems especially have performance requirements as an essential part of their correct behavior. The behavioral view should include a definition of the expected workload and the required responses of the system to this workload.

Ideally, the behavioral view should complement the functional view. Each transaction in the functional view should be traceable through the system from the initial input through the interfaces and functional units to the final output.

2.4. Classification Scheme

The classification scheme shown in Table 2-1 uses the system stages (specification, design, implementation) on one axis, and the views of the system (behavioral, functional, and structural) along the other axis. The requirements stage is **not** included, since it is informal, and the authors of this report believe there is little to be gained by including it. A high-level classification of a method can be obtained by determining if it supports the activities of a particular stage and, for each stage supported, whether or not the method provides a means to represent each of the three views. In addition, as discussed in Chapter 3, the same framework can be used to identify the operational characteristics of a system to be developed.

Views of the System	Specification	Design	Implementation
Functional			
Structural			
Behavioral			

Table 2-1: Stages of Development

3. System Characteristics

It is important to consider the characteristics of a system when choosing a method to use in its development. A method that is suitable for one system may be unsatisfactory for another. For example, real-time systems deal with the processing of data by a computer in connection with another process outside the computer according to time requirements imposed by the outside process [IEEE 83]. "Real-time systems typically sense and control external devices, respond to external events, and share processing time between multiple tasks. Processing demands are both cyclic and event driven in nature [Fairley 85]." Real-time systems are very different from, for example, batch oriented, data processing systems where the rate of data input and output is controlled by the system rather than by an external process. Methods suitable for data processing systems need not model an external process or the unique characteristics of devices that sense and effect the environment. Methods for use in the development of real-time systems must model both.

One of the first steps in evaluating a method is to characterize the operational system that is to be built and to understand how the method will help developers deal with each of its characteristics. Using the framework discussed in Chapter 2, the characteristics of an operational system can be well described by considering three views of the system: the functional view, the structural view, and the behavioral view. During the development process, the functional and behavioral views are developed and gradually transformed to the actual software represented by the structural view.

Our discussion of these views focuses on the ability of methods to represent the views, analyze the representations, and provide guidance to developers in deriving the views. General, system-independent characteristics of methods and their representations are discussed in following chapters.

Beginning with this framework, the first questions to ask when evaluating methods for requirements analysis and design are:

1. Does the method allow the representation of all three views?
2. Are the views complementary?
    - Can there be integrated views of function and behavior as well as independent views?
    - Are there suggested techniques or rules for deriving the structural view from the functional and behavioral views?

The discussion on views is divided into two major sections. First, the functional and behavioral views are discussed in the section on operational characteristics. Second, the structural view and the constraints often placed on software structure are discussed in the section on structural characteristics.

3.1. Operational Characteristics

The operational characteristics of a system primarily describe what a system does (functional view): what functions it performs, what input the system receives, what output the system gives, and the relation between the inputs and outputs. In addition, operational characteristics also address when the functions occur (behavioral view); which functions occur under which conditions, and how quickly inputs are transformed to outputs.

Each of the following subsections deals with one of these views and lists evaluative questions that should be asked of a method when considering the view.

3.1.1. Functional

The functional view shows the system as a collection of processes operating on data. This view deals with processes; data flows, including system inputs and outputs; and data stores. The functional view is discussed below in terms of the environment in which the system operates, the system's inputs and outputs, and the data transformations performed by the system's processes.

3.1.1.1. Environment

An important characteristic of any system is its relationship to the environment in which it operates. Specific systems are built to operate in specific environments and these environments differ dramatically across different types of systems. Characterizing the environment involves specifying the entities with which the system interacts: users, devices that sense and effect the environment, communications channels that connect the system to other systems.

1. Does the method provide a representation that clearly draws a boundary around the system and separates it from its environment?
    - Does the representation clearly identify the specific entities that the system interfaces with?
    - Does it provide a mechanism to detail and describe the interfaces between the system and these entities?

3.1.1.2. I/O

Another system aspect that is dealt with early in the analysis process deals with the characteristics of the system's individual data inputs and outputs. Initial analysis work often begins with a hazy picture of the inputs and outputs. In other cases, inputs and outputs are well defined and detailed since the system must interface with existing devices.

1. Does the method allow the representation of data that flows across these interfaces using an appropriate level of abstraction?
    - Are abstract representations available to describe data flows that are to be detailed by analysts and designers?
    - Are detailed representations available to describe data flows that must meet rigidly defined interfaces to existing devices or must conform to established communications protocols?

3.1.1.3. Data Transformations

Characteristics of individual data transformations (processes) performed by the system must also be considered when selecting methods. Different systems or different parts of an individual system require processes that differ in size, complexity, and dynamic behavior. Development methods must deal with the characteristics of the processes required in the system.

1. Does the method provide a technique to represent each process including its inputs, outputs, functions, and the exceptions that it may raise?
2. Does the method provide a representation (decision tables, for example) that allow analysts and designers to define and detail complex logic when required?

3.1.1.4. Math Representations of Engineering Phenomena

Many of the results of engineering studies associated with systems are represented as mathematical models of the system operation. These models may represent the kinematics of motion of relevant objects, detail the processing of received signal data, or describe the scheduling algorithms used to service asynchronous requests for scarce resources. These models are derived from engineering studies, and simulation is often used in the development and validation of the algorithms. Unfortunately, the models are often incomplete, are developed somewhat independently of the final target environment, and must be carefully incorporated into the software design. If the algorithms themselves are still under development during the software design, the method must accommodate this process and provide aids to minimize the associated risk.

1. If mathematical algorithms must be devised, does the method provide a representation that is familiar to the algorithm developers and can be understood by the algorithm implementors?
    - Do the representations allow description of the precision and accuracy or the calculations dictated by the requirements?
    - Do the representations allow specification of functionality under adverse conditions such as loss of data or single sensor failure?
2. Does the method provide techniques to refine and validate the algorithms over time?

3.1.2. Behavioral

The behavioral view shows the way the system will respond to specific inputs: what states it will adopt, and what outputs it will produce for each combination and time sequence of inputs and state transitions. The behavioral characteristics of individual systems vary considerably and an effective development method must deal with the required characteristics of the system to be developed. Behavioral characteristics are discussed below in terms of: modes and states; capacity, workload, and performance, and human interfaces.

3.1.2.1. Modes and States

Real-time systems are, to a large extent, event driven and a system's response to any particular event is dependent not only on the event itself, but also on a variety of conditions that have occurred prior to the event. A system's operational state represents an externally observable mode of behavior [Ward 85] where the system's response to an event or set of events differs

depending on the current operational mode. In addition, real-time systems often change state based on conditions detected within the system itself, for example, hardware component failure or input signal overload.

Methods used for the development of systems with these characteristics must support the analysis and specification of this type of behavior.

1. Does the method incorporate the concept of describing the behavior of the system using a state-oriented model?
2. Does the representation of the model include the representation of events, actions, states, transitions, and guards dictated by the operational environment?
3. Is the model appropriate for the complexity of the system under development?
    - Are representations such as event tables and transition tables [Fairley 85] available for simple behavioral models?
    - Are representations such as state transition diagrams or Petri-nets available to describe complex models?
    - Can the representations for complex models be partitioned to help developers deal with the complexity and the math formulations?
4. Can stimulus/response relationships be represented in a time-dependent manner?
5. Does the method allow representation of the relationship between the behavioral model and the functional model?

3.1.2.2. Capacity, Workload, and Performance

Real-time systems react to events occurring in their environment and data provided by the environment. Events occur in periodic or aperiodic intervals and data from different sources is available at distinct points in time or continually. In addition, different events may occur and need to be processed concurrently.

Capacity, workload, and performance are three related characteristics all dealing with the ability of a system to process its inputs to produce outputs over a defined period of time. The capacity of the system can be described by defining all of the entity types required in the system, the relationships between those entities, the attributes associated with each entity type and relationship, and the number of instances of each entity type. The workload is a measure of the amount of work presented to the system. Workload must be defined according to a time line so it can be analyzed frame by frame. The required performance characteristics define the constraints on processing specific elements of the workload.

Consider a display system, for example. The capacity of the system may be described by stating that there are two types of displays, tabular and graphical, with 200 tabular displays and 500 graphical displays. The tabular displays may contain up to 8 columns and 30 rows of numbers or character strings, and the graphical displays may contain up to 20 interconnected entities, with each entity and connector having up to 4 text strings or numbers describing its values. The workload of the system may state that at each of five operator stations, the operator may request up to five displays per minute. The required performance may be described by stating that a selected graphical display must be displayed completely to the operator within two seconds of

selection, and a tabular display within three seconds of selection. These response times should be adhered to when simultaneous selections at five active stations are made.

These characteristics can be specified for normal loading, the most likely load, and for stressful (exceptional) operating conditions. In addition, fallback and recovery modes must be accounted for. Building a system to meet only the normal load often leaves the human operator in a situation of dealing with the stressful conditions with little help from the automated system. On the other hand, building a system that handles all overload conditions simultaneously is unwise and very expensive. It is often true that some operations can be postponed, delayed, or ignored during such conditions.

Questions regarding the ability of a method to deal with capacity, workload, and performance are listed below.

1. Does the method allow the representation of periodic and aperiodic events?
2. Does the method allow the representation of discrete and time continuous data?
3. Does the method allow the representation of input rates and bounds on those rates?
4. Does the method allow the representation of concurrent processes?
    - Does the method allow the representation of synchronization between concurrent processes?
    - Does the method provide analysis techniques for synchronization between communicating processes, mutual exclusion for shared resources, deadlock?
5. Does the method provide a representation that captures capacity requirements for individual entities?
    - Can this representation be related to the time line so that total capacity required to meet performance constraints can be derived?
6. Does the method provide representations that capture performance requirements?
    - Does the method allow the representation of relative as well as absolute time constraints; e.g. action A occurs within two milliseconds after the occurrence of event B?
    - Does the method provide techniques to analyze these representations to determine if the performance requirements will be met under normal conditions; under all conditions?
7. Does the method assist the developer in handling exception, fallback, and recovery conditions?
8. Does the method encourage using the behavioral model to specify system operation under overload conditions?

3.1.2.3. Human Interface

An especially important characteristic of many real-time systems is the system's interface to its human operator. The human interface is significantly different from other types of system input/output interfaces and requires separate attention in any development effort. A system's success or failure often depends on its human interface characteristics. Since this is an area that is not well covered by most methods, it is discussed in more detail in Chapter 9, Problem Areas.

3.2. Structural Characteristics

A system's software is structured from a set of components and the interfaces between the components. A structural view of the system is concerned with the individual components, the interfaces between them and the distribution and flow of data and control between the components through the interfaces. As stated in [Statemate 87], the structural view is concrete and can be thought of as implementing the functional and behavioral views, which are more abstract.

A system's software structure is influenced by a variety of factors and a development method should account for these factors. Certain structural characteristics are desirable regardless of the type of system being developed and are discussed in the Chapter 4. Other characteristics are influenced by factors such as the overall system architecture and the implementation language platform, which vary across systems under development. Development methods, to be effective, should help developers deal with these system dependent characteristics.

3.2.1. System Architecture

One factor that plays a major influence on the structure of real-time software is the hardware architecture of the overall system of which the software is a part. Because of severe performance requirements, needs for extreme reliability in harsh environments and the state of existing hardware technology, hardware architectures that support software in real-time systems often have the following special characteristics: distributed processing and robustness.

3.2.1.1. Distributed Processing

Because of the need for processing power or the need for geographic distribution, systems are often constructed as networks of homogeneous or heterogeneous processors. In addition, real-time systems often contain devices that sense and effect the environment or special processors (signal processors as an example) that are customized to rapidly perform particular data transformations. These processors communicate via special purpose busses, shared memories, communications lines, or some combination of these devices. Methods, to provide a complete structural view of the system, must account for these characteristics.

1. Does the method provide representations that describe all elements of the hardware system?
2. Does the method provide representations that detail the data and signal flow between the devices?
    - Does the representation identify the source of all stimuli and provide a map to the resulting outputs?
3. Does the method provide modularization guidelines that account for the need to map specific software modules onto specific hardware devices?

3.2.1.2. Robustness

Systems have varying requirements for robustness: availability and reliability. Robustness generally is described by probabilistic measures such as mean time to failure and mean time to repair. Robustness considerations often dictate the system architecture, requiring hardware redundancy to offer protection from individual hardware component failures. The hardware

architecture characteristics dictated by the need for robustness heavily influence the software structure and the dynamics of the software's operation.

1. Does the method provide representations that describe the mapping of software components onto hardware components for all operational configurations?
2. Does the method provide techniques to detect and recover from failures?
    - Are techniques for defining and implementing error detection and correction codes provided?
    - Are techniques for implementing fault-tolerant software provided?
    - Does the method provide techniques to model dynamic system reconfiguration?
3. Does the method provide techniques to analyze system performance for all configurations?

3.2.2. Data Modeling

One of the major aspects of a system is its data. The data objects manipulated by the system, the relationships between them, and the degree of distribution of the data objects are important in structuring the system. In systems that are primarily concerned with data (for example, DBMS) there are often many and complicated relationships between the data objects, and it is important to represent the data in a consistent and complete manner. In systems with less emphasis on data, there are often fewer and less complicated relationships, and data modeling is usually less difficult.

1. Does the method provide a data modeling technique, describing all entities and their relationships?
2. Can this technique represent different views of the data and integrate these views?
3. Can the technique allow for partitioning of the representations, or must they be coalesced into a single underlying view?
4. Can complex relationships be described in straightforward ways?
5. Does the method provide representations that detail the distribution of data in distributed processing systems?

3.2.3. Language Platform

A second major factor that influences the structure of a system's software is the need to utilize "standard" operating systems and language systems in the system's implementation. These standards are often imposed with the goal of reducing the complexity, development time and life cycle cost of the system. Each language platform (language, runtime system, operating system) provides designers with a set of constructs, entities, and mechanisms that they must use to construct the system's software. Effective methods should assist developers in structuring software that utilizes these components.

1. Does the structural representation provided by the method map directly to the structures provided by the language platform?
2. Can the representation be used to detail the interfaces between the application specific software and the language platform software?

3. Do the data structure representations provided by the method map directly to the data definition forms of the language platform?
4. Do the partitioning and modularization techniques provided by the method lead to structures that can be efficiently implemented on the language platform?

4. Constraints

As stated in [Firth 87b], the task of system developers includes not only the activities of analysis, decomposition, and understanding of the systems operational requirements, but also includes the activities of analysis and understanding of the solution's constraints in order to compose a solution that operates within those constraints.

All real-time software solutions must be designed by considering a variety of constraints: hardware architecture requirements, implementation language and software platform requirements, the needs of integration and test teams, and the anticipated needs of those who will evolve the software system. Meeting these constraints usually is not a simple task and often involves many separate groups of people. Each group has its own set of requirements and needs. For example, the end-users who will use the system on a long-term basis will have a different set of requirements than the integration and test team or the software developers. Often, these needs conflict with one another, so that the needs of one group cannot be fully satisfied unless the needs of another group are compromised. Throughout the software life cycle, decisions must be made constantly and iteratively to determine the optimal software solution. A method can help this continuous evaluation of constraints and solutions in a number of ways:

- By representing the system so that all groups in the development life cycle can understand the tradeoffs involved in order for the system to meet certain constraints.
- By providing a means to segregate and identify the source of the individual constraints: customer, user, hardware availability.
- By representing the system so that all groups in the development life cycle can understand the parts of the system with which they are not involved, so that they can formulate requirements that are realistic, feasible, and work within the boundaries of existing system requirements.
- By helping developers analyze these tradeoffs to understand the impact of a particular requirement on the entire system and to evaluate alternatives in order to generate a system that meets its constraints.
- By guiding developers towards solutions with characteristics that are known to reduce the complexity and cost of development and maintenance tasks.

Several constraints were discussed in the section on structural characteristics of Chapter 3. This chapter deals with additional areas that must be considered when evaluating methods. Effective methods should support developers efforts to compose and implement an acceptable solution.

4.1. Software Architecture

It has long been recognized that a major portion of the expense of developing and using software comes from the activities of integration and test of the software and evolving the software after its initial use [Boehm 76]. Certain general characteristics of software architectures have been recognized as valuable in reducing these costs. Architectures that exhibit these characteristics tend to be easier and less expensive to implement, test, and maintain.

The various processes and data structures defined during the requirements analysis and

specification activities must be packaged in a manner that promotes testing and maintenance. Packaging involves grouping components together and defining the interfaces between the packages. The packaging process involves making many tradeoff decisions but should, to the largest extent practical, lead to an architecture that exhibits the interrelated characteristics of modularity, information hiding, and clearly identified exception handling. Detailed discussions of these characteristics can be found in a variety of texts including [DeMarco 79], [Fairley 85], and [Habermann 83]. This section lists desirable attributes of design methods that should be considered.

4.1.1. Modularity

Modularity deals with the physical composition of a system: its components and their interfaces. Each component should be well defined and of manageable size and complexity. The interfaces between the modules should also be well defined and designed to minimize the complexity of the connections between modules, thereby enhancing the clarity of the design. The concepts of size, complexity, coupling, and cohesion can be used to describe the desirable properties of modular systems.

4.1.1.1. Size and Complexity

The basic need underlying the concept of modularity is the need to partition systems into a number of separate, manageable pieces. While it may be possible for a single designer to successfully implement and maintain a small program, it is generally not possible to construct a software system as one monolithic piece. A design method should encourage partitioning a system into pieces of manageable size and complexity.

1. Does the method provide techniques to partition the system into manageable pieces?
    - Do these techniques promote the notion of limiting the size of individual modules so they can be easily understood?
    - Do these techniques promote limiting the complexity of individual modules?
2. Does the method provide a representation that describes each piece?
3. Does the method provide a representation that describes the interfaces between the pieces?

4.1.1.2. Coupling

A software system should be structured to minimize the number and complexity of connections between its modules. The term coupling is often used to describe the interdependence of modules. Systems that exhibit low coupling of modules are generally easier to test and maintain.

1. Does the method provide packaging techniques that lead to minimal coupling between modules?
    - Does the packaging technique discourage the use of common data structures?
    - Does the technique encourage the use of passed parameters?
    - Does the technique encourage the localization of control decisions?
2. Does the method's design representation enforce reference to other modules by name rather than by elements internal to the referenced modules?

4.1.1.3. Cohesion

Cohesion is a measure of how strongly the various elements of a module are associated with each other. To simplify the tasks of test and maintenance, modules should be composed of elements that are strongly associated with each other. Cohesion of elements can be described using the scale provided in [Fairley 85] where modules exhibiting the greatest cohesion are the most desirable.

1. Does the method provide a modularization technique that leads to modules of high cohesion?
    - Does the method encourage grouping together elements that are all related to performing a single function?
    - Does the method encourage grouping together elements that are related to a single real-world object?
    - Does the method encourage encapsulating major data structures in modules with procedures to manipulate the data?
2. Does the method provide rules for examining the design representation that assist developers in determining the cohesion of the modules?

4.1.2. Information Hiding

Information hiding deals with the desirable "black-box" nature of individual modules. Information hiding suggests that any program or module should be viewed in terms of its inputs and outputs only. Details of encapsulated data structures, algorithms, or the structure of the module itself should be hidden from other modules. It should not be necessary to deal with the internal details of a module in order to use it.

1. Does the method provide a modularization technique that encourages the use of information hiding?
2. Does the method provide a design representation that separates module interface information from module implementation information?

4.1.3. Exception Handling

A special area of concern in designing systems is how the system handles errors—exceptions to normal operations. Exceptions include such things as improper data values, input data rates out of defined bounds, attempts to operate on unavailable data, etc. Most exceptions must be recognized and processed by the applications software, rather than the underlying run-time system or operating system, since only the applications software "understands" the context in which the error occurred and the possible actions that can be taken to recover from the error.

Exception handling software should be clearly separated from the "normal" software. Mixing the two together significantly increases the complexity of the software and may lower its reliability.

1. Does the method emphasize the need and provide guidance in separating normal processing from exception handling?
2. Can the method's design representations be used to capture and represent this separation?
3. Does the method's behavioral model account for the need to model exceptions?

4.2. Integration and Test Constraints

During integration, the various subsystems (collections of modules with defined interfaces) that comprise the software are tested to ensure that they all work together. Integrators should be able to understand the requirements that each module or subsystem meets in order to design test suites and to ascertain whether a group of modules do indeed work together properly. Often test suites are defined in parallel with the system. Testers must understand the system characteristics and the characteristics of the environment within which the system will operate in order to develop effective test suites. In addition, since target system hardware is often not operational until late in the development cycle, initial testing is often performed on special integration systems that simulate the target system.

Integration teams and/or quality assurance teams often consist of people other than those who developed the software. Representations generated during development should thoroughly describe the function and behavior of the system as well as model the environment in which it operates so that testers can more easily identify and isolate problems.

1. Do the representations describe the intended function and behavior well enough so that separate teams can use them to test the system?
2. Does the method provide representations that model the system's environment to allow testers to develop real-world test scenarios?
3. Can the method be employed to design and document test cases?
4. Can test teams trace from requirements through the representations to develop test cases for modules and subsystems?
5. Does the method provide guidance on extrapolating the results of tests executed with target system simulators to predict actual characteristics of the software on real target hardware?

4.3. Evolution Constraints

Real-time systems are not static entities. After a system has been deployed, it will presumably operate over a long period of time. Over the course of its life software bugs and usage problems may surface. What was originally thought to satisfy the needs of the user may in fact be insufficient, or different sets of users may evolve different sets of needs. In addition, over time, the environment within which the system operates may change, thus affecting the system requirements and constraints. For example, the system may need to meet new performance or reliability requirements. This may require that enhancements be made to the software or that new hardware technologies be used. The software must be able to change to conform to the needs and constraints that arise over the course of a system's life.

1. Do the methods representations provide maintainers with a "road map" into the implementation that provides an overview of the system, shows the relationship of its parts, and allows them to focus quickly on areas of interest?
2. Do the representations help maintainers determine the scope of effect of a proposed change to a particular module or set of modules?
3. Does the method provide a modularization technique that partitions hardware-dependent and hardware-independent functions into separate modules?

4. Does the modularization technique lead to architectures that accommodate small changes to the system's timing requirements without major redesign?
5. Does the method promote the notion of abstraction of hardware device functions into logical operations to support the replacement of devices over time?
6. Does the method provide techniques for organizing its representations to support the evolution of the system into multiple versions?

5. Representations

Any evaluation of a method must focus on how well the method can be used by system engineers, analysts, and designers to assist them with the complex tasks of requirements analysis, and specification and design. During the development process the system under development must be modeled from a behavioral, functional, and structural view. These models must capture and record the various characteristics of the system. They should, in addition, exhibit the characteristics described below.

5.1. Abstraction

"The essence of abstraction is to extract essential properties while omitting inessential details [Ross 75]." Abstraction allows the designer to concentrate on each representation, at various levels of detail. The designer can more easily think about complex problems in a hierarchical manner by abstracting away from the details: thinking about the problem at a high level first, and then proceeding on to the next lower level. This may also make finding problems with the representations easier, as problems concerning a particular piece of data can be isolated at one level. In addition, a designer may also be able to define one high-level representation for similar objects, and differentiate those objects at a lower level of the representation. Abstraction supports the notions of information hiding [Parnas 72]. With information hiding, details that do not affect the system at a particular level of abstraction are made inaccessible. Representations at any given level have access to the bare minimum of information that they need to describe the system at that level of abstraction. A representation can then be examined at a level that is appropriate for the reader, so that he does not need to be concerned with details that he does not need to know about.

1. Does the method define abstraction techniques and give guidance on producing representations at various levels of abstraction?
    - Are data abstraction techniques available to define and represent abstract data types?
    - Are representations available to detail these data types?
    - Are procedural abstraction techniques encouraged?
    - Does the method supply definitions for commonly used procedural types, e.g. stacks, queues, bounded buffers?
    - Can the user define and record the definition of abstract procedural types of special use for his applications domain?
2. Do the abstraction techniques include the definition of balancing rules?
    - Are developers given rules to use to insure consistency between the levels of abstraction?

5.2. Consistency

The method must help the designer make sure that consistency is maintained between representations so that "no set of individual requirements is in conflict" [IEEE 83]. The representations should all be consistent with the requirements of the system, and the method should help the designer determine that this is true. In addition, the various types of representation (structural, functional, behavioral) should be derived and expressed in a manner that encourages consistency between them; that is, they should not contradict each other. The method should prescribe heuristics to examine the representations for consistency, such as semantic and syntactic analysis, simulation, and by tracing components back to representations that have already proven to be consistent.

1. Does the method provide guidelines for analyzing representations to insure consistency within each representation?
2. Are representations expressed in a manner that allows for consistency checking between them?
3. Does the method encourage the use of a common glossary or dictionary to protect against naming clashes between entities?
4. If hierarchical representations are available, does the method encourage the technique of using one level of representation to derive a template for the next lower level?

5.3. Completeness

A representation is complete if all the necessary elements that describe the system are included. Completeness, of course, is difficult to formally define and it is difficult to check to see if a representation is in fact complete. However, the designer needs a mechanism to determine that all aspects of the system have been represented completely. The method can help by forcing or encouraging the designer to consider a range of issues when deriving the representations.

- Does the method provide mechanisms to represent, examine or understand such things as exceptional conditions, boundary conditions, error handling, initialization, fault tolerance, performance, and resource constraints?
- Does the method provide a mechanism to ensure that all of the requirements for the system have been met?

5.4. Complexity

A careful distinction must be made between a useful and expressive representation and one that is cluttered, complicated, and difficult to read and understand. Representations should not be more complex than the nature of the relationships that they are trying to express. Each representation should express only a few key concepts and relationships. A cluttered diagram sometimes indicates that not enough abstraction has occurred or that too many things are being expressed at once. In addition, although formal notations are able to express information in a concise manner, they sometimes can be elaborate and cumbersome to use. This is compounded if the method uses a number of notations that are syntactically and semantically different.

1. Are there a manageable number of concepts expressed in a single representation?
2. Does the method provide techniques to partition and decompose complex representations into sets of simpler representations?
3. Are notations semantically and syntactically simple across representations, and are the semantics and syntax relatively simple and straightforward to use?

5.5. Traceability

A method should guide the designer in deriving representations at each stage of the development process, as well as between the levels of each representation within a stage. It should be easy for the designer or reader to understand and trace through these representations. A designer must be able to refer to representations that he has done previously, so he may use what he has done to move forward to the next stage or level in the design. The designer also may need to look backwards to understand the repercussions of a change at any level. Similarly, others involved with the software will need to be able to trace backwards and forwards between the stages and levels of representation in order to verify requirements or to understand the system. Reusers of software components need traceable representations, for often it is not sufficient for a designer to simply reuse an existing implementation (a portion of code); the requirements and design of the component must be included to help the designer understand the function and behavior of the component.

1. Can readers of the method's representations easily determine the paths between requirements and implementation?
2. Does the method provide naming conventions for entities across all representations?
3. Does the method provide notation for relating the name of an entity with the names of its components?
4. Does each level of a hierarchical representation clearly identify its parent and children?
5. Does the method encourage recording and provide representations to record the designers critical decisions, e.g., which processes and data stores have been pulled together into which packages; which packages model real-world objects?
6. Can a time ordered sequence of events be traced through the representations to determine the behavior of the system?

5.6. View Integration

A method should allow the designer to specify relationships between representations at different levels or stages. This way the designer can narrow the problem initially, for example, concentrating on the behavioral representation of the system and later relating the functional and structural representations to finish the description of the system. Because the representations are integrated, the designer may examine what effects a functional change in the system will have upon its behavior. While representations should be coupled, there should not be a large amount of redundancy between them.

1. Does the method provide techniques for relating one view of the system to another?
2. Do the techniques prescribe a small number of well defined relationships?
3. Do the method's representations clearly identify the relationships?
4. Can developers use the representations to determine how alternatives in one would effect the others?

5.7. Ambiguity

Completed representations should be clear, precise, and complete to avoid ambiguity. The designer may at any stage leave portions of the system without representations, however, in the end these ambiguities must be resolved. The method should allow the designer to derive representations, leaving some areas temporarily ambiguous. In addition, the form of the final representation must be well defined so that misinterpretation of the representation is not possible.

1. Does the method prescribe a sequence of steps that allows developers to leave portions of a representation temporarily incomplete and ambiguous?
2. Does the method provide techniques for examining the representation to insure all ambiguities have been resolved?
3. Does the method provide representations that are well defined, e.g., do decision tables contain entries for all combinations of conditions?
4. Can various audiences examine the representations to gain an unambiguous understanding of the system at the level of detail they are interested in?

5.8. Duplication

Representations should contain a minimum of duplicated information. Information duplicated between representations often leads to inconsistent system designs, as well as making it confusing to check for completeness. Certainly, in some instances there will be a need to redundantly express information, but those instances should be few and carefully chosen. When duplication is necessary, the method should help developers identify when it occurs.

1. Can you derive representations without expressing the same information over and over again?
2. Does the method provide a representation that can be used to record the existence and location of duplicated information; e.g. cross reference table?

5.9. Changes

At any stage in the development process, one should be able to make changes to the representation of the system in a simple manner. This allows the designer the flexibility to refine, prune and enhance the design, as well as to evaluate different design alternatives. "The development methodology can assist this evolutionary activity by providing accurate external and internal system documentation, a well structured software system that is easily prototyped and modified by those making the system changes [Freeman 83]." It should be easy to add to, delete

from, or alter aspects within a particular representation as well as relationships between representations. The magnitude of change at one level of a representation should be reflected by changes of similar magnitude at lower levels or between other representations. The designer should be able trace the effects that these changes might have on other representations.

1. Does the method provide hierarchical forms for all types of representation?
    - Can low-level changes be made without necessarily effecting high-level representations?
    - Can high-level changes be made without effecting all lower level representations?
2. Does the method provide structuring techniques where the scope of effect of any decision is minimized?
3. Can the repercussions of a change to higher and lower levels of a representation be traced; across different representations?

5.10. Compliance with Standards

Use of software methods should lead to representations that comply with the applicable standards wherever it is specified or appropriate. These standards can vary from dictated languages (Ada) to company, customer, and project imposed standards used for such things as: intermediate work products used for status and progress reviews; final documentation delivered with the product; and project defined work products that compensate for known gaps in the standard development process. In support of this, the following questions should be asked of a method:

1. Does the method establish and enforce well defined representation standards for each view at each stage of development?
2. Can the method be tailored to support organizational (customer, contractor, project) standards, procedures, and paradigms?
3. If the programming language is Ada, does the method deal with the structures of Ada explicitly?
4. If the development process is to follow the 2167 standard, does the method produce artifacts (in a timely manner) to satisfy these requirements?

6. Deriving Representations

The representations provided by a method serve as useful vehicles to organize, record, analyze, and communicate the requirements and design of a system under development. Requirements analysis and design are not, however, simply a matter of writing things in fixed format and looking at the results. Deciding what to include in the representations and how to organize them are difficult tasks that cannot be reduced to mechanical transcription. Methods should, then, provide system developers with more than a set of representations. They should also provide a set of suggested heuristics or techniques to guide the process of deriving the representations.

Existing methods vary widely in the guidance they provide in deriving representations. Some prescribe a set of well defined activities that are to be performed as a regimented sequence of steps. Others offer loose collections of suggestions for producing and analyzing the representations. While rigidly defined activities seem appealing on the surface, it is generally true that the development of a real-time system is a multifaceted task that does not lend itself to a lockstep approach. The complexity of many systems and the constraints on the implementation provide developers with problems that must be solved with the skill and judgement developed through training and experience.

There are, however, techniques that have been effectively used in the development of real-time systems. Methods should remind developers of these techniques and offer suggestions on how to use them. Several of these have been discussed in previous sections since their use can be demonstrated by examining the representations, e.g., modularity discussed in Chapter 4 and abstraction discussed in Chapter 5. Others are discussed in Chapter 9, Problem Areas, since existing methods tend to be stronger in representations than in techniques for deriving them. The remaining few are described in this section.

6.1. Partitioning

Partitioning is the activity of breaking a representation of a system into a set of smaller representations while defining the interfaces between them. Partitioning helps the developer by allowing him to transform large, complex problems into a set of smaller, manageable problems. Partitioning is a critical part of requirements analysis and design since partitioning decisions made early in the process have strong effect on all later activities. For example, early decisions on how to break up a system into subsystems effect not only the architecture of the final implementation but also the basic manner in which developers view and think about the system under development.

1. Does the method provide a means to partition the system at all stages?
2. Do the partitioning techniques account for the structure of the problem as well as the hard implementation constraints?
    - Do requirements partitioning techniques lead to structures that can be understood by system users in terms that are familiar to them?
    - Do design partitioning techniques account for distribution, robustness and the need to utilize existing software packages, e.g., database management systems?

3. Are the suggested techniques flexible enough to allow experienced developers to examine the alternatives they believe are appropriate?
4. Do the techniques emphasize the need to define the interfaces between the partitions?
5. Do the techniques suggest partitioning the system so that pieces are independent enough to allow individual analysts/designers to elaborate and refine the pieces independently?

6.2. Refinement

Refinement is the process of including more detail in a representation. Usually, the developer does this hierarchically, with the most abstract level at the top of the hierarchy and each lower level including subsequently more detail. The method should guide the developer in determining the amount of refinement that should be included at each level of a representation. Each level should contain an amount of detail that can be easily understood and managed by the designer and should be consistent with the amount of detail that is contained in other representations at the same level.

1. Does the method guide the designer in determining the amount of detail to include at each level of a representation?
2. Is the amount of detail consistent across levels of different representations?

6.3. Elaboration

Elaboration of a representation occurs when developers add more functionality to a partial representation. Developers should be able to partially derive a representation, leaving some areas of the representation unfinished, and elaborate it at a later time. The developer should be able to use the same heuristics to derive the unfinished part of the representation so that it is consistent with the previously completed parts of the representation. The method should guide the developer to insure omitted parts of the design are eventually completed so the requirements of the system are met.

1. Can the developer leave parts of the representation of the system temporarily incomplete?
2. Does the method help the developer determine what representations are missing in order to satisfy system requirements?
3. Does the method provide a clear set of heuristics so that a designer can elaborate on a representation in a way that is consistent with the previously completed representation?

6.4. Reuse

Reuse of existing software with known characteristics and high reliability promises to lower system development costs, reduce risk, and yield systems of higher reliability. Experienced designers will reuse, either directly or by salvage through rework, components from another system if possible, since this is often a productive path to take.

A development method should account for the benefits and risks of reuse. While it is desirable to reuse existing components when possible, they should only be reused when they in fact help produce a system that meets the end users' needs. Focusing on reuse too early in the development process may lead to systems that are well structured and reliable but do not exhibit the operational characteristics required by the user.

Since existing requirements analysis and design methods typically ignore the area of reuse, this topic is also covered in Chapter 9, Problem Areas.

1. Does the method offer guidelines that encourage both a top down and bottom up (using existing components) approach?
2. Does the method encourage separating requirements into essential and negotiable classes, where negotiable classes identify the opportunity for reuse?
3. Does the method encourage designers to suggest alternatives provided by existing components, to negotiable requirements?

6.5. Evaluation of Alternatives

At any point in the development process, developers continually make decisions that will affect the system. Usually, there is no single right way of doing things. Alternatives must be evaluated to determine what effects a decision will have on the overall system and what tradeoffs must be made. Development methods should encourage the generation of alternatives and provide guidance to evaluate them wherever possible.

1. Does the method allow the designer some flexibility when making design decisions?
2. Does the method encourage the designer to generate a number of alternative designs?
3. Does the method help the designer evaluate alternative representations based upon system characteristics and constraints?

7. Examining Representations

It is important that a method emphasize the need and support the activity of examining its representations at all stages of development. Successful development efforts must include reviews of intermediate work products to identify problems, reduce risk and insure the delivered system exhibits its required operational characteristics. The general rule that should be followed during a development effort is to "examine early and often."

Examinations should be considered from two points of view: the goals of the examinations and the techniques used to perform examinations. Each of these viewpoints is discussed separately below.

7.1. Examination Goals

Examinations of the evolving system's representations occur at various points in time for a variety of reasons. Examinations are effective only if they are conducted for specific, well defined purposes. Current practice too often conducts examinations (walkthroughs, inspections, etc.) with no clear, unambiguous understanding of why the examinations are being done. The end result is a process that produces little of value and much frustration for all participants.

Several of the reasons for conducting examinations are discussed below. Development methods should facilitate these examinations.

7.1.1. Feasibility

A development approach is ultimately proven feasible if the system's final implementation conforms to the specification, including all of the performance and resource constraints. Examinations for feasibility should begin early in the development effort and identify major risk areas— areas where desired functionality cannot be provided at all or not within the implementation constraints. Feasibility should also be gauged throughout the development cycle.

Methods should emphasize the need to determine feasibility and should promote techniques that identify development approaches that are unlikely to be successful. Typically, it is better to compose a system from parts with known characteristics, since this makes the judgement of feasibility easier than the case where the characteristics of all parts are "guesstimated." Feasibility is also more easily predicted if the method supports the paradigms of: prototyping, incremental development of high-risk items, reusable components, techniques to assist with estimation, and prediction of resource usage and performance.

1. Does the method provide techniques to examine its representations, including operational prototypes, to assess risk and gauge feasibility?
2. Do the method's design representations allow capture of the important characteristics of existing components (performance, capacity, resource usage)?
3. Does the method provide techniques to predict the resource requirements and performance characteristics of the composition of the components?
4. Does the method encourage identification of high risk items and their incremental development?

7.1.2. Conformance

The goal of any development effort is to produce a system that conforms to its specified behavior. The representations produced during the development process must be periodically examined to determine if they capture the requirements and to determine that the evolving design and implementation will yield a product that conforms to the specification. Finally, the implemented system must be tested to determine if the actual behavior, under test conditions, matches the expected behavior.

1. Does use of the method lead to a specification that clearly and completely defines the desired operational characteristics of the system under development.
2. Does the specification serve as a model of the system that can be understood by the customer and end user to insure the system under development will meet their needs?
3. Does the method help determine what questions should be asked during the examinations for conformance?
4. Does the method provide guidance in developing test cases by specifying which tests should be developed?

7.1.3. Safety

Real-time computer systems are now embedded in systems that can cause hazards to the public at large, to the system operators, and to the equipment being operated. It is desirable that the development methods promote safety analysis. Hazardous operations often arise from poorly defined operating conditions. Safety analysis usually tries to pinpoint those conditions in a specification and design. In order to perform a reasonable safety analysis, one needs to understand how the software, hardware, human operators, and system's environment interact.

1. Do the method's representations describe the operation of the complete system, including software, hardware, human operators, and environmental conditions?
2. Does the method provide techniques to inspect the behavior of the system under exceptional conditions in the environment?
3. Does the method provide specific guidelines examining for safe operations, e.g., determining if the human operator is warned if he is taking actions leading to hazardous conditions?

7.2. Examination Techniques

Several examination techniques have proven useful in the development of systems. These techniques include: walkthroughs, inspections, analysis, test and data extraction, reduction, and analysis. Methods should facilitate these well known techniques through the characteristics of their representations and by providing guidelines for conducting the examinations.

7.2.1. Walkthroughs and Inspections

Walkthroughs can be used at various stages of the development cycle to examine the specification, design, or implementation artifacts. Walkthroughs are typically conducted by having the creator of a particular artifact "walk other interested parties through" it. The participants play the role of critical reviewers and attempt to find errors or areas of special concern.

Inspections are similar to walkthroughs in that people other than the creator of the artifact examine it to find errors. Inspections tend to be more formal in that the artifact is distributed ahead of time for review with the hope that reviewers will spend time examining it in detail. The members of inspection teams are carefully chosen by the skills they bring to the effort; different team members may inspect for different things. Finally, inspection meetings are conducted to "debrief" the inspection team members rather than to "generally discuss" the artifact.

Inspections and walkthroughs should be supported by sets of rules that allow clear communication and promote focused discussion. The rules should attempt to define the terms "error" and "area of special concern." They should also encourage reviewers to separate issues of form and substance, that is, concerns over what a representation says should be separated from how the representation says it. Errors in substance are the most important.

1. Is the syntax for each representation clearly defined so reviewers can quickly locate and dispense with problems with form?
2. Does the method provide specific guidelines for reviewing the representations?
    - Are there checklists and well defined procedures for areas such as consistency, completeness?
    - Are their issues lists for areas such as exception handling, robustness, performance, maintainability?
    - Are criteria for determining the quality of a representation, unique to the method, clearly spelled out?

7.2.2. Analysis

Analysis is a technique for assessing, against a fixed set of criteria, certain aspects of a representation. Analysis can be used for any representation but is typically used with source code or design representations. There are two general classes of analysis: static and dynamic. Static analysis is concerned with the structural characteristics of the representation, i.e., is the representation well formed by the structural rules provided by the method.

Dynamic analysis is concerned with the behavior of a representation when it is "executed." Testing is a form of dynamic analysis performed against code. The notion of "executable specifications" recognizes the need to perform dynamic analysis against other forms of representations earlier in the development process.

1. Does the method provide techniques and a clear set of rules for static analysis of its representations?
2. Is the syntax of the representations well defined allowing the purchase or development of automated tools to perform static analysis?

3. Does the method support the animation or simulation of its representations to allow dynamic analysis early in the development cycle?

7.2.3. Testing

Testing is the form of dynamic analysis most commonly used and understood. Testing involves executing the software system on a set of well defined test data. The purpose of testing is to insure that the individual parts of the software system do what they were designed to do and that the composition of the parts meets the system's operational requirements. While testing, unfortunately, cannot prove the absence of errors, it can be used to improve one's confidence in the system.

1. Can the specification be used to develop test scenarios that exercise the system under both normal and exceptional operating conditions?
    - Does the method suggest a technique for generating the test scenarios?
    - Does the technique explain how to use the representations to insure all important aspects of the system are covered?
2. Does the method provide a technique for using the design representation to generate unit test cases?
3. Does the method provide a technique for using the behavioral representation to generate behavioral tests?

7.2.4. Data Extraction, Reduction, and Analysis

One aspect of analysis generally overlooked in discussions of development is the need to perform analysis of a system after it has been put into use. Once a system is installed, it often has residual flaws that have escaped detection in the tests performed. These flaws are usually the result of unusual operating conditions, including race conditions and unexpected failure conditions. The side effect of these flaws can vary from causing the system to "crash" to minor errors in operation. In any case, there is a need to analyze the system's behavior to isolate the cause of the problem. This is best done by anticipating that the system will have these flaws and designing the system to continually collect data on its own operation. Data is generally extracted from the system in a raw format, reduced to a well defined format, and analyzed to determine the cause of the observed errors.

1. Does the method encourage the designer to define the data necessary for extraction?
2. Does it encourage the use of its behavioral model to identify high-risk areas (areas of concurrency, synchronization, potential race conditions) where data collection would be especially useful?

8. Management Characteristics

A software system is developed by a software "manufacturer" who will deliver the software product for profit to the customer. "The need to control cost in all its forms places the software engineering process—and those who conduct it—in a set of managed activities that are frequently reviewed and adjusted by those responsible for the cost of system production, system acquisition and ownership [Firth 87b]." Each group of actors in the software development cycle imposes constraints on the process. The software builders need to perform risk assessment, plan, organize, staff and track the development of the software system. For efficiency's sake, they usually have mechanisms existing within the organization to perform these activities. Similarly, customers may also specify reviews, tests, and deliverables in order to ensure that the software product possesses the form and content that they desire and to ensure that costs are being kept in line. The mechanisms and procedures that a customer or developer has in place to monitor and regulate the software process have a significant impact on the use of any software development method. "A software development methodology is actually a blend between a collection of technical procedures and a set of management techniques that can result in effective deployment of project personnel, predictability of project schedule, budget and outcome, accurate estimation of software properties, and the final result of a high-quality system that meets the needs of its users throughout the lifetime of the system [Freeman 82]."

8.1. Process

Most organizations who develop software have "in-house" standards and practices to which all software development projects adhere. The same can be be said for customer organizations that contract the development of software. There are specified activities involved with planning, organizing and staffing, project tracking and control, and risk assessment that must performed and specified criteria to be met in order for a software project to gain management approval to advance a step forward in the development cycle. Usually, these standards and practices have been formulated over time and stem from the need for management to control the costs and output of the project in a way that serves the best interest of the company and the customer. Often, standards and practices have evolved from lessons learned on previous projects, and, thus, management is firmly committed to this way of doing business, as they may have previously paid a high price for problems that occurred before these practices were implemented.

Because organizations implement these process controls for a reason, and because management usually believes strongly in these controls, a software development method should help support efforts by management to control the software development process. However, there is usually a dichotomy in the way that management and development processes are driven. Management processes are usually time driven. The customer negotiates to have the product delivered on a particular date, and managers try to schedule the development process in order to meet specified checkpoints—milestones and deliverables that lead up to delivery. However, the software development process is event driven. The completion of one step is usually necessary for the commencement of another. Modules should be completed and tested before integration can begin. Requirements analysis should be completed before high-level design is begun. The

method should augment the management practices that are already in place and should not require severe alterations in an organization's process of developing software. Both the method and the management process should be flexible enough to allow for synchronization of the software development process and management checkpoints. It won't be efficient to schedule reviews and walkthroughs of products that are not yet completed, nor would it be wise to deliver faulty or unfinished products to the customer.

A software development method can provide managers with a series of activities and steps on which they can base schedules and monitor their progress, representations which can be used to document decisions made during development or as deliverables, and rules to analyze representations to support reviews and to judge how complete or correct a group of representations are.

1. Does the method provide planning techniques that lead to milestone definitions and project plans that are consistent with use of the method and the implementation language?
2. Does the method have analysis techniques that can be used during reviews or that can help you gauge progress during development?
3. Are representations clear and easy enough to understand to be used for design reviews?
4. Can representations be used to comprise deliverables?
5. Can you rapidly develop high-level design representations that can be analyzed to determine the most feasible design approach?
6. Does the method prescribe the generation of a sufficient number of intermediate design products to support a detailed project plan?
7. Does the method help partition the system into manageable pieces that can be given out to individuals?
8. Does the method contain rules that review teams can use to analyze or verify the design?

8.2. Cost

The direct cost of using a method may appear to be small - a few books or training courses might seem to suffice. However there are many other costs associated with successfully introducing and using a method in your organization. In order for a method to be worth using, the initial "learning curve" costs and other associated costs should be more than offset by the other benefits a method provides for your software project.

One significant cost is time. It will take time to train your people to adequately use a method and to have them adjust to using it. Some methods have elaborate notations and require extensive training, while others are somewhat simpler to use. Your organization must determine whether time spent for training and extended schedules is feasible. It may take more time, especially in the beginning, for your team to execute the project, given the extra activities that a method may prescribe and their unfamiliarity with the method. It may also take time for management to understand how to best incorporate the use of a development method into their existing process.

Another cost of acquiring a method is the cost of evaluating and selecting automated tools that support it. A good set of automated tools can increase the efficiency and ease of using a method by a large amount, however they must be selected carefully [Firth 87a].

1. Are there a number of automated tools on the market that support the method?
2. Does the method require a reasonable amount of training, that is proportionate to its power, and feasibility of use?
3. Does use of the method reduce product life cycle costs by an amount that is worth the cost of adopting the method?
4. Can the method be adapted for use: across a broad range of applications domains, with a variety of implementation languages, with conformance to a variety of standards?

9. Problem Areas

One of the necessary activities in evaluating existing methods is to gain an understanding of not only what a particular method is but also what that method is not. The software engineering methods that exist today can aid the solution of a variety of problems. It is important to recognize, however, that they are not "silver bullets" [Brooks 87] that deal perfectly well with all problems, handle all aspects of system development, eliminate the need for experienced personnel, or guarantee success under any conditions.

Successful selection and use of methods require a realistic understanding of their capabilities. False expectations about the benefits of using a method will undermine its successful use and may cause an individual or an organization to abandon the use of methods entirely. As stated in [Firth 87b], it is important to determine a method's undesirable features.

1. What issues does the method not deal with?
2. What are the negative consequences of using the method to solve a particular problem?

Method assessors and users need to answer these questions and to develop a realistic plan for introducing the method into the development organization. This plan should address the problem areas and, for each problem, should describe the way in which the method will be augmented to deal with each of the problems.

The following sections discuss problems that have been identified, in general, in current methods and their use. Several of the areas below have been briefly discussed in previous sections but are grouped together here to present a more complete picture of problems in existing methods. While it may be possible to identify a particular method that deals especially well with one of the difficult areas below, this is the exception rather than the rule. Also, methods that focus their attention on one of these areas usually offer little support in others.

9.1. Experienced Personnel

It is important for the investigator and evaluator of methods to remember that methods are techniques used by developers to produce software systems. Methods, by themselves, do not produce specifications, designs, or implementations. They provide guidance, forms of representation and rules for analysis that assist developers in the production and analysis of a variety of intermediate and final products. Methods are not a substitute for a development staff that is skilled and experienced in developing systems within a particular applications domain. Knowledge and understanding of an applications area, as well as experience in developing systems in that area, remain crucial to the success of any development effort. When used properly by an organization, methods can help leverage the skills of experienced practitioners and improve the overall productivity of the organization and the quality of its products.

Evaluative questions, then, should focus on how experienced practitioners can use a particular method to effectively capture, represent, and communicate the key characteristics of a system

that only they, through their experience, would recognize as important. The questions should also focus on how less experienced staff members can use the method to receive structured, efficient guidance from those with more experience.

1. Can experienced staff members use the method to capture and represent what they believe to be the key functional, behavioral, and structural characteristics of the system at all stages of development?

    - Can they effectively use high-level representations to explain these characteristics and the decisions they have made about them to the variety of audiences involved in the development process?
    - Can they use these representations as a vehicle to direct the work of less experienced staff members?
    - Can they use the analysis techniques prescribed by the method to test all of their thinking and answer their questions?

2. Can less experienced staff members use the rules and guidelines prescribed by the method to develop elaborations and refinements of the high-level representations?

    - Can they use the examination techniques prescribed by the method to insure that their elaborations and refinements are consistent with the higher level representations.
    - Can they use the derived representations to support structured dialogues with more experienced staff members to uncover gaps in their own understanding and skills?

9.2. Transformation Across Stages

The development process, consisting of the specification, design, and implementation stages, must generally be tailored by individual organizations for specific product development efforts. Often, during the process, representations for each stage of development are partially complete, inconsistent, and ambiguous. This situation is recognized by emerging life-cycle models such as the spiral model [Boehm 86]. It is motivated by the need for developers to initially tackle perceived high-risk issues and drive them through design and occasional implementation to insure the viability of the design approach.

Given this need, it is especially helpful if a method deals with all stages of development and does so in a way that can be tailored to the problem at hand. Partial or complete representations at one stage should support the development of representations for the next stage. All representations should remain useful throughout the development effort and should capture the history of key design decisions.

Unfortunately, this complete set of characteristics is lacking in any one method. Many cover only a subset of the phases, provide little guidance in transforming from one set of representations to the next, or require the developer to dismember one set of representations to produce the next. Evaluative questions should focus on the breadth of representation, ability to tailor the sequence of stages, and the transformation from one stage to the next.

1. Can developers use the method to represent the system under development at all stages?

2. Can developers use the method when narrowing in on high risk areas to derive partial representations for all stages?

    - Does the method support the partitioning of the problem into a set of smaller problems with well defined interfaces?
    - Do the analysis rules support the examination of these fragments with the same rigor as they support the examination of complete representations?
    - If the method allows for the simulation or animation of the specification, does it equally well support the simulation or animation of the fragments?

3. Are developers, using the method, supported by a set of transformation rules or guidelines that allow them to transform the representations at one stage to those of the next?

    - Do the rules prescribe a transformation process that is relatively automatic?
    - Do the entities and structures created at one stage of development remain visible and intact at the next stage?
    - Does a representation continue to serve a useful purpose after the representation for the next stage is completed?

9.3. Feasibility Analysis

Feasibility analysis deals with assessing the capability of a system's design and implementation to conform to its specification, including all of the performance and resource constraints. In most real-time systems development efforts it is important to begin gauging the feasibility of a design or design fragments early in the design process before great time and effort is spent going down futile paths. The most difficult problems in this area typically deal with the behavioral characteristics of workload, capacity, and performance discussed in Section 2.1. A useful method must not only capture these behavioral requirements, but it must also assist the developers in determining whether or not a developing design will meet the requirements. Evaluative questions should focus on the ability of a method to support performance prediction.

1. Can designers use the method to predict the resource requirements and performance characteristics of an evolving design?

    - Do the method's design representations capture resource estimates for individual components?
    - Do the same representations capture performance estimates for these components?
    - Does the method support execution or animation of the design representation and predict resource usage and performance under varying operating conditions?

9.4. Development Constraints

The application specific software for many real-time systems is often constrained to operate on specific hardware, runtime system and language platforms. The use of these platforms is dictated by a variety of technological, organizational, and business issues that are treated more fully in [Fairley 85], [Freeman 83], and [McDonald 85]. In addition, an engineering organization may own or be able to acquire existing software components that have been used and thoroughly tested in other systems. In an attempt to reduce development time, improve reliability and lower life cycle costs, developers may be required to reuse many of these components. The overall result of these constraints is that the application software must be constructed using the constructs, entities, and mechanisms provided by the implementation platform and existing components.

Additional constraints, often overlooked in discussions of methods, are constraints placed on the development process. Most development groups must adhere to "company standard" or "customer standard" project planning, tracking, and control mechanisms. A method can only be effectively used if its use can somehow be integrated with project management schemes.

Many existing methods do not account for these issues. They either do not provide a bridge to transform the specification to an acceptable architecture or they fail to fully account for the operational characteristics of the implementation platform and components that must be used. Project management concerns are often overlooked in method selection.

Evaluative questions dealing with development constraints are:

1. How well can the method be used to create reusable components?
    - Does the method encourage partitioning into standard parts?
    - Does the method incorporate an abstraction mechanism that allows development of generic components?
    - Does the method provide a design representation that captures component characteristics important for reuse: inputs, outputs, exceptions, capacity, performance, restrictions?
2. Does the method include a bottom-up approach?
    - Can developers represent existing components and the implementation platform?
    - Do the representations capture performance characteristics and resource consumption of each component?
    - Can the method predict the resource requirements and performance characteristics of the composition.
3. Can the method be used with existing project management practices?
    - Can the method’s representations be used to fulfill project milestones and deliverables?
    - Do the method’s examination rules and techniques fulfill the customer’s progress review requirements?

9.5. Large-Scale Problems

The evaluation of alternative methods must deal with the ability of these methods to "scale up" to support very large development efforts. Real-time systems vary considerably in size and a method that deals well with small problems is not necessarily suitable for large problems.

Evaluative questions should deal with the ability of the method to partition the problem into manageable pieces, integrate those pieces as the solution develops, and control the derivation of the various representations.

1. Can developers use the method to partition the problem into a set of smaller problems with well defined interfaces and integrate the results?
    - Does the method provide hierarchical representations that allow developers to work at a detailed level and easily remember the overall context?
    - Are the rules used for partitioning and decomposing one view consistent with the rules for other views? For example, if the method supports data flow and control flow, are they dealt with at compatible levels?
    - Does the method endorse naming conventions that allow multiple developers to integrate their results and avoid naming clashes?
    - Alternatively, does the method embody the notion of scope?
2. Does the method deal with the issue of identifying the representations themselves?
    - Are the representations uniquely identified in a manner that mirrors their relationships?
    - Does the method provide rules for configuration control?
    - Does the method provide rules for version control?

9.6. User Interface

A critical aspect of most systems, often overlooked by existing methods, is the system's interface to the user. In many real-time systems— avionics and air traffic control, for example—the user, pilot or air traffic controller must be viewed as a key element of the overall system. The user's needs must be met and the needs must be satisfied in a manner that is acceptable to the user. A system that performs all its real-time functions properly but fails to meet the special information processing and interaction needs of its users will not be successful in meeting its overall operational goals.

In addition, a system's user interface is one area that is likely to change as the system is developed and after it has become operable. It is often the case that users redefine and refine their own understanding of their needs only after they have had the opportunity to interact with the system being developed. Also, as hardware technology evolves, there is often a strong demand to replace the hardware devices the user deals with directly. Newer devices can often ease the burden on the user or provide information displays that are easier for the user to comprehend.

Evaluative questions should focus on the ability of the method to deal with human factors as well as the guidance the method provides in architecturally separating user interface components.

1. Does the method provide representations that allow the user to visualize the user interface?
    - Do the representations adequately represent what the user will see, content and format of displays?
    - Do they allow representation of user inputs, content and format?
    - Do they allow the user to visualize a dialogue with the system, the ability of the user to modify the displays and the mechanics of the interactions?
2. Does the method model and help predict the information processing and decision-making demands placed on the user?
3. Does the method promote the partitioning of user interface processing from other processing?
    - Does it encourage a clear separation between form and substance of user interactions?
    - Does the design method address user error processing; are lexical, syntactic, and semantic errors dealt with at an appropriate distance from the interface?
    - Does it encourage the separation of device-dependent from device-independent processes?

9.7. Implementing Designs

Partitioning a development process into various stages is done to separate a variety of issues and to allow different sets of people with different sets of skills to effectively participate in the process. Design and implementation activities are separated for a variety of reasons, but there must be some transformation process between design and implementation representations. Creating a design that cannot be feasibly implemented on a constrained implementation platform serves no useful purpose.

An example of the problem of transforming designs into implementations that is of special interest to DoD contractors is their need to implement systems using the Ada language. The Ada language and Ada runtime systems are in the process of maturing as users are maturing in their ability to effectively use both. Many existing methods were developed prior to the introduction of Ada and do not necessarily embody or emphasize the same underlying development principles that are promoted by Ada. Runtime systems for target hardware have seen little use in deployed systems and their operational characteristics are not yet widely understood. In addition, many designers and implementors have little experience with Ada implementations. Taking all this together it is especially important to consider the aspect of Ada maturity when selecting methods.

Evaluative questions should pay attention to implementation issues when choosing a design method that will be used in conjunction with an Ada implementation.

1. Do the method's representations and techniques for deriving the representations emphasize the importance and use of the concepts underlying the Ada language:

- abstraction
- information hiding
- concurrency
- modularity and packaging
- exception handling

2. Does the method provide a design technique that results in design representations that map easily to Ada implementations?
    - Does the method provide a data modeling technique that results in representations that map directly to Ada supported data types?
    - Does the method provide a modularization technique that results in representations that map directly onto packages and tasks?
3. Does the design method encourage the use of idioms, commonly used procedural types, that are especially difficult to implement in Ada?
4. Does the design method lead to architectures that will not (when implemented using a particular compiler or runtime system) meet required performance or resource usage constraints? For example, does the method lead developers toward implementing systems as collections of communicating tasks while the required implementation platform is unable to efficiently support task switching or is unable to support a large number of tasks with minimal resources?

10. Conclusion

The guidelines that are presented in this report can be used to help an organization assess a method using the five step process outlined in [Firth 87b]. Again, the five steps are:

1. Determine the important characteristics of the system to be developed and determine how individual methods help developers deal with those characteristics.
2. Identify the constraints imposed on the permitted solutions and determine how individual methods help developers deal with those constraints.
3. Determine the general usage characteristics of the individual methods. A method can be examined by developing an understanding of how it represents a system under development, the guidelines it gives developers to derive the representations, and the guidelines it provides to examine the representations.
4. Determine the support that the method provides to those who must manage the development process as well as the costs of adopting and using a method.
5. Develop an understanding of the issues that the methods do not address.

The steps place a strong focus on understanding the system that will be built, including potential risk and problem areas, as well as understanding the environment within which the system will be developed. This understanding is best achieved by defining a manageable sample problem that is representative of the system and applying likely methods to the solution of the problem. By using this approach, method assessors and potential users will gain an in-depth understanding of the methods and their strengths and weaknesses.

There is no such thing as an overall "best" method for developing all software, only the method that will work best to help develop a system with particular characteristics and will blend with an organization's software development practices. This may require tailoring the method to existing practices, modifying existing practices to account for effective techniques new to the organization, or both.

In fact, the characteristics of a good development method are not much different than the characteristics of good software engineering. A method is a tool that can help you employ good software engineering practices. Even so, choosing a good method will not shield you from failure. Just like any other tool that promotes good software engineering practice, a good method can fail if it is used poorly or incorrectly, or used in place of experienced, competent people.

By applying the evaluative questions in this document to existing methods and by reviewing Chapter 9 (Problem Areas), it should become apparent that existing methods do not provide solutions to all of the problems encountered in the process of developing software systems. Those choosing methods must recognize the need to anticipate how the method will be used by individuals in their organization to solve particular problems and the need these individuals will have to augment the method with additional techniques to cover the areas the method does not address. Methods should not be introduced into organizations as panaceas for all problems. False expectations on the benefits of using methods can cloud their real value and lead to their unjustified abandonment. Problem areas and techniques for dealing with the problems should be addressed in an introduction plan.

Finally, a cost/benefit analysis should be performed to predict whether adopting a particular method will pay off in terms of reducing costs and improving the quality of the delivered product. Acquiring proficiency in the use of a method requires training and experience over time. Also, a tool that automates your chosen method can enhance the benefits obtained from its use, but these benefits do not come without a price (see [Firth 87a]).

While it is easy to estimate certain costs (books, classes, seminars, tools), it is not easy to estimate others (time and experience to become proficient). Adopting new methods often requires changing the way people view and think about problems. The cost of bringing about this change and the benefit received from it are not easy to predict. Selecting and introducing a method must be followed by carefully monitoring its use to determine the success its users are having in improving productivity and the quality of the delivered software product.

Acknowledgement

The authors are grateful to the many people who reviewed early drafts of this document and provided constructive comments and criticisms. We are especially thankful for the in-depth review and comments provided by Len Bass, Timothy Coddington, Richard D'Ippolito, Jeffrey Stewart, and Nelson Weiderman of the Software Engineering Institute as well as by Cammie Donaldson of Software Productivity Solutions, Inc., Don Reifer of Reifer Consultants, Inc., and Hank Stuebing of the Naval Air Development Center.

References

[Alford 85] Alford, Mack.
SREM at the Age of Eight: The Distributed Computing Design System.
Computer 18(4):36-46, April, 1985.

[Boehm 76] Boehm, Barry.
Software Engineering.
IEEE Transactions on Computers C-25(12), December, 1976.

[Boehm 86] Boehm, B. W.
A Spiral Model of Software Development and Enhancement.
ACM SIGSOFT Software Engineering Notes 11(4), August, 1986.

[Booch 83] Freeman, P., and Wasserman, A. (editors).
Object-Oriented Design.
In Tutorial: Software Design Techniques, IEEE Computer Society Press, Washington, DC, 1983.

[Brooks 87] Brooks, F. P.
The Silver Bullet, Essence and Accidents of Software Engineering.
IEEE Computer 20(4), April, 1987.

[Cameron 83] Cameron, J.
Tutorial: JSP and JSD: The Jackson Approach to Software Development.
IEEE Computer Society Press, Washington, DC, 1983.

[DeMarco 79] DeMarco, Tom.
Structured Analysis and System Specification.
Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, 1979.

[Fairley 85] Fairley, Richard E.
Software Engineering Concepts.
McGraw Hill, New York, 1985.

[Firth 87a] Firth et al.
A Guide to the Classification and Assessment of Software Engineering Tools
Carnegie Mellon University, Pittsburgh, PA, 1987.

[Firth 87b] Firth et al.
A Classification Scheme for Software Development Methods.
Technical Report CMU/SEI-87-TR-41, ESD-RT-87-204, Carnegie-Mellon University, Software Engineering Institute, November, 1987.

[Freeman 83] Freeman, P., and Wasserman, A. I.
Ada Methodologies: Concepts and Requirements.
ACM SIGSOFT Software Engineering Notes , 1983.

[Gomaa 86] Gomaa, H.
Software Development of Real-Time Systems.
Communications ACM 29(7):657-8, July, 1986.

[Habermann 83] Habermann, A. N. and Perry, Dewayne E.
Ada for Experienced Programmers.
Addison-Wesley Publishing Co., Reading, Massachusetts, 1983.

[Harel 86] Harel, David.
Statecharts: A Visual Approach to Complex Systems.
Concurrent Systems , February, 1986.

[IEEE 83] The Institute of Electrical and Electronics Engineers, Inc.
IEEE Standard Glossary of Software Engineering Terminology.
Technical Report ANSI/IEEE Std. 729-1983, IEEE, February, 1983.

[McDonald 85] McDonald, Catherine W.; Riddle, William; and Youngblut, Christine.
STARS Methodology Area Summary - Vol II: Preliminary Views on the Software Life Cycle and Methodology Selection.
Prepared for Office of the Undersecretary of Defense for Research and Engineering, IDA Paper P-1814.
March, 1985

[Parnas 72] Parnas, D. L.
On the Criteria To Be Used in Decomposing Systems Into Modules.
Communications ACM 15(12):1053-1058, December, 1972.

[Ross 75] Ross, D. T., Goodenough, J. B., and Irvine, C. A.
Software Engineering: Process, Principles, and Goals.
Computer , May, 1975.

[Statemate 87] AdCad.
The Languages of Statemate1.
January, 1987

[Teichrow 77] Teichrow, Daniel.
PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems.
Transaction Software Engineering SE-3(1), January, 1977.

[Ward 85] Ward, Paul T. and Mellor, Stephen J.
Structured Development for Real-Time Systems, Vol I: Introduction & Tools.
Yourdon Press, New York, 1985.

[Webster 81] .
Webster's Third New International Dictionary.
Merriam-Webster, Inc., Springfield, MA, 1981.