IN-61-CR
312538
p22

(NASA-CR-187400) A METHOD FOR TAILORING THE INFORMATION CONTENT OF A SOFTWARE PROCESS MODEL (Houston Univ.) 22 p CSCL 09B

N91-13097

Unclas
G3/61 0312538

A Method for Tailoring the Information Content of a Software Process Model

Sharon Perkins
University of Houston-Clear Lake

Mark B. Arend
University of Houston-Clear Lake

June 1990

Cooperative Agreement NCC 9-16
Research Activity SE.8

NASA Johnson Space Center
Engineering Directorate

Research Institute for Computing and Information Systems
University of Houston - Clear Lake

T·E·C·H·N·I·C·A·L R·E·P·O·R·T

The RICIS Concept

The University of Houston-Clear Lake established the Research Institute for Computing and Information systems in 1986 to encourage NASA Johnson Space Center and local industry to actively support research in the computing and information sciences. As part of this endeavor, UH-Clear Lake proposed a partnership with JSC to jointly define and manage an integrated program of research in advanced data processing technology needed for JSC's main missions, including administrative, engineering and science responsibilities. JSC agreed and entered into a three-year cooperative agreement with UH-Clear Lake beginning in May, 1986, to jointly plan and execute such research through RICIS. Additionally, under Cooperative Agreement NCC 9-16, computing and educational facilities are shared by the two institutions to conduct the research.

The mission of RICIS is to conduct, coordinate and disseminate research on computing and information systems among researchers, sponsors and users from UH-Clear Lake, NASA/JSC, and other research organizations. Within UH-Clear Lake, the mission is being implemented through interdisciplinary involvement of faculty and students from each of the four schools: Business, Education, Human Sciences and Humanities, and Natural and Applied Sciences.

Other research organizations are involved via the "gateway" concept. UH-Clear Lake establishes relationships with other universities and research organizations, having common research interests, to provide additional sources of expertise to conduct needed research.

A major role of RICIS is to find the best match of sponsors, researchers and research objectives to advance knowledge in the computing and information sciences. Working jointly with NASA/JSC, RICIS advises on research needs, recommends principals for conducting the research, provides technical and administrative support to coordinate the research, and integrates technical results into the cooperative goals of UH-Clear Lake and NASA/JSC.

A Method for Tailoring the Information Content of a software Process Model

Preface

This research was conducted under the auspices of the Research Institute for Computing and Information Systems by Dr. Sharon Perkins, Associate Professor of Applied Science at the University of Houston-Clear Lake, and Mark B. Arend, of UHCL. Dr. Perkins also served as RICIS technical representative for this activity.

Funding has been provided by the Engineering Directorate, NASA/JSC through Cooperative Agreement NCC 9-16 between NASA Johnson Space Center and the University of Houston-Clear Lake. The NASA technical monitor for this activity was Dave Howes, Information Systems Manager, Engineering Directorate, NASA/JSC.

A Method for Tailoring the Information Content of a Software Process Model

Dr. Sharon Perkins
University of Houston, Clear Lake

Mark B. Arend
839 Walbrook Dr.
Houston, TX 77062

ABSTRACT

This paper will define the framework of a general method for selecting a necessary and sufficient subset of a general software life cycle's information products, to support new software development projects. Procedures for characterizing problem domains in general and mapping to a tailored set of life cycle processes and products will be given. An overview of the method is shown using the following steps:

1. During the problem concept definition phase, *perform standardized interviews and dialogs* between developer and user, and between developer and customer.
2. *Generate a quality needs profile* of the software to be developed, based on information gathered in step 1.
3. *Translate the quality needs profile* into a profile of quality criteria that must be met by the software to satisfy the quality needs.
4. *Map the quality criteria* to a set of accepted processes and products for achieving each criterion.
5. *Select the information products* which match or support the accepted processes and product of step 4.
6. *Select the design methodology* which produces the information products selected in step 5.

This paper will address every step, but will not attempt to generate a full-up methodology. A few of the more popular process models and design methodologies known today will be examined for their information content.

TERMINOLOGY NOTES

The terms "software process model" and "life cycle" will be used interchangeably. The term "user" will always mean "customer and user".

INTRODUCTION

The complete set of information products defined for common software process models and development methodologies is often too large for certain development efforts. In many cases, a subset of information products and the activities that produce them will suffice to administer the development of a software product. The act of selecting appropriate information products and activities to support the development effort is called "tailoring" the life cycle or development methodology. This tailoring process is currently an ad hoc method performed by managers and developers, in early meetings with the customer and user, as they begin to define some sort of Software Management or Development Plan. This paper explores a more formalized tailoring method to assist in the definition of such plans. It is hoped that such a formalization will both speed the process and help ensure the selection of a necessary and sufficient subset of information products (and by implication, the activities which produce them).

The cornerstone of this tailoring method uses Software Quality Assurance (SQA) techniques. Traditionally, SQA has dealt with the detection and prevention of defective software. New ideas in the field of SQA are concentrating on beginning the function much earlier in the life cycle, as early as problem concept and initial requirements definition. It is hoped that SQA principles will assist the user and developer in creating complete, consistent and

This paper developed under NASA cooperative agreement NCC-9-16, through the RICIS office of the University of Houston, Clear Lake. April 1990.

testable requirements. This assistance offers guidelines up front when we're scrambling to put some sensible words on paper.

I believe that two quotes [5], [21] can illustrate the idea of "engineering in" quality to a software product.

You can't achieve Quality...
unless you specify it!

Quality must be defined as conformance
to requirements, not as "goodness"

USING SQA TECHNIQUES TO SPECIFY QUALITY

Quality Factors

This is a common SQA term. **Quality Factors** are characteristics which a software product exhibits that reflect the degree of acceptability of the product to the user. Since we're moving SQA up front, we'll restate this: Quality Factors are characteristics which the user *requires* the software to exhibit in order to reflect *the best possible* degree of acceptability.

Table 1 shows a list of Quality Factors which has been coming into general use for some time [21]. It was first proposed at the Rome Air Development Center (RADC) in 1977. I show a slightly expanded list, as it has evolved somewhat since then [5].

There are more detailed meanings of the quality factors which guide the user & developer in determining how important each factor is for their application.

Not every project requires all quality factors, which is good, because some quality factors are at conflicting purpose. Shown below is a list of factors whose characteristics cause conflicts of definition.

Quality Factor Conflict	*Explanation of conflict*
Efficiency vs. Integrity	Overhead required to control access negates efficiency.
Efficiency vs. Usability	Overhead required to ease operations negates efficiency.
Efficiency vs. Maintainability	Optimized code negates maintainability. Modularization, instrumentation and well commented high-level code increases overhead.
Efficiency vs. Testability	Optimized code negates testability.
Efficiency vs. Portability	Optimized code is dependent on host processor services.
Efficiency vs. Flexibility	Overhead required to support flexibility negates efficiency.
Efficiency vs. Reusability	Overhead required to support reusability negates efficiency.
Efficiency vs Interoperability	Overhead required to support interoperability negates efficiency.
Integrity vs. Flexibility	Flexibility requires general and flexible data structures, increasing data security problems.
Integrity vs. Reusability	Generality required by reusable software introduces protection problems.
Integrity vs. Interoperability	Coupled systems allow more avenues of access.
Reusability vs. Reliability	Generality required by reusable software increases difficulty of providing error tolerance (anomaly management) and accuracy.

The conflicts shown do not mean that the two factors are in strict mutual exclusion -- extra effort may be expended to address the difficulties of specifying factors in conflict. Note that efficiency tends to conflict with

Quality Factor	Meaning of factor in context of user needs for software product
Correctness	Conformance of software design and implementation to stated requirements.
Efficiency	Economy of resources needed to provide the required functionality.
Expandability	Ease of maintaining the software to meet new functional or performance requirements.
Flexibility	Ease of maintaining the software to work in environments other than originally required.
Integrity	Security against unauthorized access to programs and data.
Interoperability	Ease of coupling the software with software in other systems or applications.
Maintainability	Ease of finding and fixing errors.
Manageability	Ease of administrating development, maintenance and operation of the software.
Portability	Ease of maintaining the software to execute on a processor or operating system other than that originally required.
Usability	Ease of learning & using the software, and of preparing input & interpreting output.
Reliability	The rate of failures in the software that render it unusable.
Reusability	Suitability of software modules for use in other applications.
Safety	Protection against loss of life or liability or damage to property.
Survivability	Continuity of reliable execution in the presence of a system failure.
Verifiability (testability)	Ease of verification of functionality against requirements.

Table 1 - Quality Factors

many other factors. This is due to the tradeoff with the additional overhead required to satisfy other quality factors that does not necessarily apply to the algorithm's basic function. Efficiency issues may also be resolved by judicious hardware selection. Note that there is also a reverse-matrix of quality factors (not shown) that tend to support one another, such as testability and maintainability -- similar sets of criteria support both factors.

So you get the idea of defining quality needs for specific applications. As this process of definition continues, a *profile* begins to emerge that describes the proposed software in terms of weighted quality factors.

The Quality Profile

I introduce this term to describe the prioritized, weighted list of quality factors that the user & developer define for their software development effort. The Quality Profile is a "signature" or "fingerprint" of a project's quality needs. Humphrey [10] offers a common-sense example of what kinds of factors are important for different applications, based upon the "primary concern" of the application.

	Primary Concern	High Priority Quality Factors
a.	Effect on human lives	Reliability, Correctness, Testability
b.	Long life Cycle	Maintainability, Flexibility, Portability
c.	Real time application	Efficiency, Reliability, Correctness
d.	In-house tool	Efficiency, Reliability, Correctness
e.	Classified Information	Integrity
f.	Communicating systems	Interoperability

The High Priority Quality Factors shown for each type of application begin to define that application's quality profile. The profile of an application of type "a" is given by high degrees of reliability, correctness and testability, and lower degrees of the remaining factors. In practice, we define a more precise scale of degrees and assign a particular weight to each factor. The resultant set of quality factor weights defines the quality profile for the proposed software.

Another example, more generic, is given by Deutsch [5] to suggest an initial prioritization of Quality Factors by "software category".

	Software Category	High Priority Quality Factors
a.	Critical	Safety, Survivability, Correctness, Maintainability, Efficiency
b.	Support	Maintainability, Verifiability, Interoperability, Portability, Usability, Correctness
c.	I/O	Correctness, Interoperability, Maintainability
d.	Data	Interoperability, Portability, Reusability
e.	Computational	Correctness, Maintainability
f.	Environment	Maintainability, Verifiability, Correctness, Interoperability, Portability, Reusability, Efficiency, Integrity
g.	MMI	Integrity, Usability
h.	Documentation	Correctness, Maintainability
i.	Design	Expandability, Flexibility, Interoperability, Maintainability, Portability, Reusability, Verifiability

These two examples offer starting points for the development of a Quality Profile. Many applications will exhibit multiple concerns or cover several categories. It is the job of the user & developer to define the Quality Profile for the specific application.

Defining the Quality Profile

Deutsch [5] suggests a metric for ranking or weighting quality factors.

Level of quality required		What techniques should be used to ensure a quality factor of this rank
E	Excellent	Exceptional techniques
G	Good	Better than average techniques
A	Average	Normal corporate practices
NI	Not an Issue	No special techniques

He then extends the metric into the realm of cost and schedule prediction, using Jensen and COCOMO model relative cost and relative schedule analysis factors. Cost and schedule prediction will not be pursued further here.

Latter day SQA is also developing standardized means by which the user and developer discuss and come to an agreement of these factors for each application. These means often take the form of questionnaires that prompt the user to evaluate all needs for quality.

Quality Criteria

This is a common SQA term. **Quality Criteria** are detailed subcharacteristics which the software exhibits that reflect the degree to which the Quality Factors are present. In other words, the planned presence of high-level quality *factors* implies the presence of a detailed set of quality *criteria*.

The Quality Factors are user-oriented; they are designed to map easily to a user's needs for the proposed software. The Quality Criteria are more software-oriented; they are designed to map easily to characteristics that may be evaluated by direct testing of the software. The relationship between quality factors and quality criteria is analogous to that between the two common stages of requirements definition. The analogy does *not* apply to the amount of effort needed to go from the early phase to the later -- Quality Factors may be translated immediately to Quality Criteria. Table 2 shows a list of Quality Criteria [5], [21].

Quality Criterion	Meaning of criterion in context of software product
Accuracy	Achievement of required precision in calculations and outputs
Anomaly Mgmt	Behavior for recovery from failures
Augmentability	Maintenance effort required to expand upon functions and data
Autonomy	Degree of decoupling from execution environment
Commonality	Use of standards to match "look and feel" of similar applications
Communicativeness	Appropriateness of inputs and outputs
Completeness	Degree to which all software is necessary and sufficient
Conciseness	Amount of code used to implement algorithm
Consistency	Use of standards to achieve uniformity within software
Distributivity	Physical (device) separation of function and data (addresses backup)
Document Quality	Access to complete, understandable information
Communication Efficiency	Usage of communication resources
Processing Efficiency	Usage of processing resources
Storage Efficiency	Usage of storage resources
Functional Scope	Range of applicability of software product's functions
Generality	Range of applicability of software's internal units
Independence	Degree of decoupling from support environment
Instrumentation	Amount of code devoted to usage measurement or error identification
Modularity	Cohesion & Coupling of software's modules (design & code)
Operability	Ease of operating the software
Safety Management	Degree to which the design addresses hazard avoidance
Self-Descriptiveness	Understandability of design & code
Simplicity	Degree to which algorithms map to the problem they solve
Support	Functionality that addresses the administration of maintenance
System Accessibility	Controlled access to functions, data and instructions
System Compatibility	Use of standards to match interfaces with hardware & communications
Traceability	Ease of finding links between requirements, design and code
Training	Provisions to help users learn the operation of the software
Virtuality	Separation of logical implementation from physical component
Visibility	Objectivity of evidence of correct functioning -- ease of test verification

Table 2 - Quality Criteria

Mapping Quality Factors to Quality Criteria

There is a direct translation from each Quality Factor to a subset of Quality Criteria which support the factor. The sets of criteria that support different factors may be disjoint or may intersect. Some criteria exhibit conflicts similar to those examined for quality factors. Table 3 shows a translation between Quality Factors and Quality Criteria that shows how the criteria support and influence the factors, either positively or negatively. The traditional

Quality Criteria \ Quality Factors	Correctness	Efficiency	Expandability	Flexibility	Integrity	Interoperability	Maintaianability	Manageability	Portability	Reliability	Reusability	Safety	Survivability	Usability	Verifiability
Accuracy		-								++		+		+	
Anomaly Mgmt	+	-								++		+	+	+	
Augmentability		--	++	+							+				
Autonomy		-		+		+			++		+		+		
Commonality					-	++					+				
Communicativeness		-		+			+				+			+	+
Completeness	++									+				+	
Conciseness	+	+					++								+
Consistency	++			+			++			+	+				+
Distributivity												+	+		
Document Quality								++			++				
Communication Efficiency		++							-						
Proccesing Efficiency		++							--						
Storage Efficiency		++							--						-
Functional Scope						+					+				
Generality		--	+	++	-	+				-	++				
Independence		-		+		++			++		++				
Instrumentation		-					+							+	+
Modularity		-	+	++		++	++		++		++		+		++
Operability		-									+			++	
Safety Mgmt												++			
Self-Descriptiveness		-	++	++			++		+		++				++
Simplicity	+	+	+	++			++		+	++	++				++
Support								++	+		++				++
System Access Control		-		-	++	--						+		+	
System Compatibility		-													
Traceability	++			+			+				+				++
Training											+			++	
Virtuality			+										+		
Visibility							+								++

Table 3 - Quality Factors <=> Quality Criteria Map

direction of translation is from criteria to factor -- the SQA or test team measures the criteria from the software, and reports on what quality factors the software thus exhibits. Our method will begin with the user definition of quality factors, and develop a set of criteria that the software must meet in order to satisfy our quality needs.

This table is merged from two different authors' approach to the factor/criteria map [5], [21]. Their perspectives overlap to a high degree, but each one shows a few more, different criteria than the other. I have included them all here in order to work with the most complete universe of factors and criteria possible. Detailed examination of the authors' text reveals that while some factors and criteria sound very similar, they actually do describe different characteristics of the software.

Symbols are used in the cells of the matrix in Table 3 to indicate the influence a criterion has on various factors. Another viewpoint is that they indicate which criteria are necessary to support each factor. A plus under a factor means that the software should be required to exhibit the corresponding criterion, but is subject to trade-off based on any conflicts that arise. A double plus means that the criterion is more important, and less subject to trade-off. A negative under a factor means that it would be wise not to require the software to exhibit the corresponding criterion, but is subject to trade-off based on the influence of other factors. A double negative means that extra effort must be expended to require the software to exhibit the corresponding criterion.

The assignment of pluses and minuses is a subjective process, but the concept has been refined over time by various authors [5], [8], [10], [21].

SOFTWARE PROCESS MODELS

"The software process is the technical and management framework established for applying tools, methods and people to the software task" [10].

There are a handful of well-defined "process models" or "life-cycles" in the industry today. They each describe a set of activities and products designed to support the successful creation of a software product. The most widely used model is called the Waterfall model. Other models are coming into use that attempt to address the shortcomings of the Waterfall, but they tend to generate very similar information products. Appendix D offers a brief description of other common process models.

The Waterfall model is characterized by a *linear* set of activities and products such that each activity uses the output of previous activities as its input. Here we list general names of the primary technical products of a waterfall model.

Activity (phase)	*Major products generated by activity (phase)*
Concept Definition	Feasibility Study, Concept document
User Req. Definition	Level-A Requirements Document, Software Management Plan, System Interface Control Document (ICD)
System Req. Definition	Level-B Requirements Document, Subsystem ICDs
System Design	System Design Document, System Test Plan
Implementation	Software, Test Case Document
Testing	Test Report
Maintenance	Upgraded Software, Maintenance Report

Note that the waterfall model itself does not really define details of the information products that are to be produced. Most users of the waterfall model recommend a larger set of documentation; these recommendations are usually laid out in a documentation standard.

SOFTWARE DOCUMENTATION STANDARDS

A Documentation Standard defines all information products that may be generated to support development of the software product. Usually, a documentation standard is packaged with a life-cycle standard. Two common standards are:

SMAP Information System Life Cycle & Documentation Standards [15]

DOD-STD-2167A [6]

For this study, we will use the document set defined by NASA's Information System Life Cycle Documentation Standard -- Appendix A shows the complete list. Our tailoring method will address which of these products are most important for a given set of quality factors.

ANALYSIS & DESIGN METHODOLOGIES

Within the framework of the software process model, some method must be used to define the content of each product. Formalized methodologies address the complex definition of the requirements and design products of the software process. There are many different methodologies to choose from for use within any software process. The information content of the requirements document, then, may vary according to the technique used to produce it.

For example, one may choose to specify system requirements using:

a. a simple textual notation developed in an ad hoc manner, or from lessons learned during prototyping.
b. a functional decomposition hierarchy of diagrams, capturing the requirements in processes and data flows.
c. an information model, capturing the requirements in objects, relations and behavior diagrams.
d. a viewpoint/behavior model, capturing requirements in data/action maps and state diagrams.
e. a hybrid of the above techniques, or other techniques.

Appendix C gives a brief overview of some of the more popular methodologies in use today, and lists all the specific products they offer. Our tailoring method may eventually be used to select a meaningful subset of these products; the current version of the paper will not explore this.

TAILORING INFORMATION PRODUCTS

The hierarchy of SMAP-recommended information products for the software development effort is shown in Figure 1.

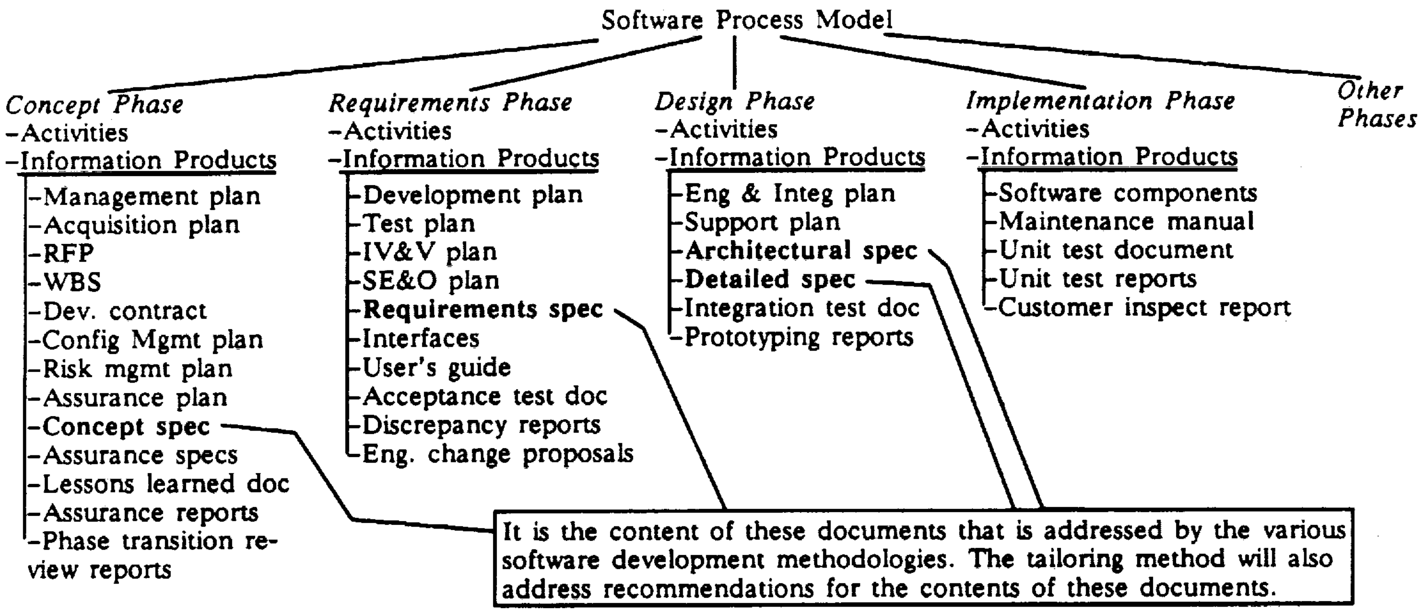

Figure 1 - SMAP Information Product Overview

Each Information Product shown will be analyzed to determine which quality criteria it best supports. The same analysis will be applied to the information products generated by various development methodologies. At this point, we will be ready to translate a set of 15 user defined Quality Factors into a recommended set of information products.

Tailoring will proceed on three levels:

1. A subset of the document universe will be selected for the specific quality profile. Example: recommend producing a Software Requirements Spec, among other documents.
2. For each selected information product, a subset of it's maximum table of contents will be selected. Example: recommend defining a Data Definition section in the Software Requirements Spec, among other sections.
3. For each recommendation from the table of contents, a set of suggestions will be given to characterize the nature of the information that should appear therein. Example: make the following recommendations for

the contents of the Data Definition section: minimize the number of different data representations, minimize number of data conversions, use dynamic memory allocation, pack all data items, etc.

The user/developer then examine the lists of recommendations, and decide whether they make sense in the context of the project. There may still be some manual tailoring to do, but the bulk of the job will have been performed by this method.

FUTURE WORK

The length of this study was not great enough to develop the full translation from Quality Criteria to Information Products. As a starting point, the requirements volume contents in Appendix B have been mapped to quality criteria. Areas that need more work are:

1. Develop the complete translation between Quality Criteria and all information products listed in the Appendices. This will include not only the selection of specific products, but recommendations for the character of that product's content.
2. Extend the tailoring method to include the tailoring of Management and Assurance activity products, as well as technical development products.
3. Define a weighting scheme for ranking Quality Factors that is consistent with Software Process Model and Design Methodology characteristics.
4. Analyze the list of information products generated by the outstanding process models in use today, and annotate with descriptions of the information content of each product. These descriptions should be compatible with the weighting scheme defined in area 3.

Appendix A
LIFE CYCLE PHASES & INFORMATION PRODUCTS: NASA'S SOFTWARE ACQUISITION STANDARD

This appendix lists the life cycle phases and information products for NASA's Software Acquisition Life Cycle as defined by the agency's Software Management and Assurance Program (SMAP). This set of documentation will serve as the universe from which a tailored set will be extracted.

The SMAP plan for volume roll-out describes a mechanism which allows the manager/developer to create information products as sections of one volume, or as separate individual volumes, or as a combination, depending upon the required complexity and management of the particular information product. The tailoring method will select a subset of these information products by recommending the "complexity" of each information product. It is recognized that there are considerations for tailoring other than the quality profile, especially as apply to the Management Plan. Initial tailoring guidelines will focus on the Product Specification, then the Assurance Specification.

Life Cycle Phases

Concept Definition Phase (**CD**)
Requirements Definition Phase (**Req**): User requirements, System Requirements
Design Phase: Software Architectural Design (**SAD**), Software Detailed Design (**SDD**)
Implementation Phase (**Impl**)
Integration and Test Phase: Integration & Unit Test (**I&T**), Acceptance Test (**AT**)
Maintenance, or Sustaining Engineering & Operations (**SE&O**)

Information Products: Data Item Descriptions (DID)

Management Activity Products: the Management Plan

Product	*Phase(s) during which product is generated, including updates.*

Component Management Plan	CD					I&T		SE&O
Component Acquisition Plan	CD							
Request for Proposal	CD							
Work Breakdown Structure	CD							
Software Development Contract	CD							
Configuration Management Plan	CD	Req						
Risk Management Plan	CD							
Assurance Plan	CD	Req	SAD					
Component Development Plan		Req						
Test Plan		Req	SAD					
Validation & Verification Plan		Req	SAD					
Sustaining Engineering & Operations Plan		Req				I&T		
Engineering and Integration Plan			SAD	SDD	Impl			
Product Support Plan			SAD					

Technical (Development) Activity Products: the Software Product Specification

Product	*Phase(s) during which product is generated, including updates.*							
Concept Document	CD							
Software Requirements Spec (Level-A)	CD							SE&O
Software Requirements Spec (Level-B)		Req						SE&O
External Interface Requirements		Req						SE&O
User's Guide		Req			Impl	I&T	AT	SE&O
Software Architectural Design Spec			SAD					SE&O
Software Detailed Design Spec				SDD				SE&O
Software Component					Impl	I&T	AT	SE&O
Software Maintenance Manual					Impl			SE&O
Version Description Document						I&T	AT	SE&O

Assurance Activity Products: the Assurance Specification

Product	*Phase(s) during which product is generated, including updates.*							
Assurance Specs	CD						AT	SE&O
Acceptance Test Document		Req	SAD	SDD	Impl	I&T	AT	
Integration Test Document			SAD		Impl	I&T		
Unit Test Document					Impl			

Management Control & Status Reporting Activity Products

Product	*Phase(s) during which product is generated, including updates.*							
Lessons-Learned Document	CD	Req	SAD	SDD	Impl	I&T	AT	
Assurance Reports	CD	Req	SAD	SDD	Impl	I&T	AT	
Phase Transition Review Reports	CD	Req	SAD	SDD	Impl	I&T	AT	
Discrepancy Reports		Req	SAD	SDD	Impl	I&T	AT	SE&O
Engineering Change Proposals		Req	SAD	SDD	Impl	I&T	AT	SE&O
Prototyping Reports			SAD					
Unit Test Reports					Impl			
Customer Inspection Reports					Impl			
Integration Test Reports						I&T		
Certification Reports							AT	
Performance/Metrics Reports						I&T	AT	SE&O

Appendix B INFORMATION CONTENT of the NASA-SMAP STANDARD SOFTWARE PRODUCT SPECIFICATION

This appendix lists the full table of contents for SMAP's Software Product Specification (SMAP-DID-P000-SW). This document package contains a Software Concept Document, a Software Requirements Spec, a Software Architectural Design Spec, a Software Detailed Design Spec, a delivery Version Description, a User's Manual and a Maintenance Manual. (from [15]). The contents have been extended to include a more complete list of information items that may be useful (from [1]). The extended items are italicized.

An initial pass at mapping document sections to quality criteria has been performed for the Requirements Volume -- the map uses abbreviations shown in the key below, and should be read "backwards" for each criterion. In other words, the map is to be used by selecting those document sections that show a reference to each criterion that is specified by the quality profile.

Ac:	Accuracy	DQ:	Document Quality	Sf:	Safety Management
AM:	Anomaly Mgmt	EC:	Communication Efficiency	Sd:	Self-descriptiveness
Ag:	Augmentability	EP:	Processing Efficiency	Sm:	Simplicity
At:	Autonomy	ES:	Storage Efficiency	Sp:	Support
Cm:	Commonality	FS:	Functional Scope	SA:	System Accessibility
Cc:	Communicativeness	Gn:	Generality	SC:	System Compatibility
Cp:	Completeness	Ip:	Independence	Tc:	Traceability
Cn:	Conciseness	Is:	Instrumentation	Tr:	Training
Cs:	Consistency	Md:	Modularity	Vr:	Virtuality
Ds:	Distributivity	Op:	Operability	Vs:	Visibility

Key: Quality Criteria Abbreviations

The Introduction and Related Documentation sections are recommended in their entirety for every software development effort. Content of the volumes following will be addressed by the tailoring method. (At present, only the Requirements Volume is addressed).

Introduction
- Identification of Volume
- Scope of Volume
- Purpose and Objectives of Volume
- Volume Status and Schedule
- Volume Organization and Roll-Out

Related Documentation
- Parent Documents
- Applicable Documents
- Information Documents

Concept Volume
- Definition of Software
  - Purpose and Scope
  - Goals and Objectives
  - Description
  - Policies
  - *Anticipated Uses of System*

Optional Configurations	
User Definition	
Overview of the User Organization	
Logical organization	
Physical organization	
Temporal organization	
reporting cycles	
scheduled events	
Information flow organization	
Capabilities and Characteristics	
Sample Operational Scenarios	
Anticipated Operational Strategy	
System ownership	
System administration	
operational control	
modification policy	
change support	
User administration	
departments	
skill levels	
Funding strategy	
Currently Used Procedures	
Requirements Volume	
Requirements Approach and Tradeoffs	DQ, Tc
Design Standards to be used	Cm, Cs, Md, SC
World Model (Information model) type A	Ag, Cc, Md, Sd, Vr
Entity-Relation summary (Data Requirements)	
Entities: description, attributes, class size	
Attributes: description, values, defaults, constraints, class size, retention/archive requirements	
Relationships: description, size, components, constraints	
Individuals (instantiations of entities)	
World Model (Information model) type B	Ag, Cc, Md, Sd, Vr
Objects: description, allowed operations, class size	
Allowed Operations: constructors, interrogators, iterators, etc.	
Messages: sent, received	
External Interface Requirements	Cc, EC, SC
Operational Resources & Resource Limitations	EC, EP, ES, Vr
Requirements Specification	
Process and Data Requirements	
Function Input data & Source	Ac, Ag, AM, Cc, Cm, Gn, SC, Sd, Tc, Vs
Function Transactions and Algorithms	Ac, Ag, AM, Cp, Cs, EP, FS, Gn, Md
Function Output data & Destination	Ac, Ag, AM, Cc, Cm, Gn, SC, Sd, Tc, Vs
Function Triggering mechanisms & conditions	AM, Cm, EP
Function Termination mechanisms & conditions	AM, Cm, EP
Function Expected demand	EP
Data Definition	Ac, Ag, At
Data Relationships	Ac, Ag, At
Data Protection requirements	Op
Data Validity check requirements	Ac, AM, Gn, Ip, Op, SA
Data Parameterization requirements	Ac, Ag, Gn, Sd, Vr
Data Format or Implementation Restrictions	Ac, Ag, At
System Behavior Requirements	

Phases & Modes	Ac, Ag, AM, Sf
System Actions	Ag, AM, Cm, Sf
Performance and Quality Engineering Requirements	
Timing & Sizing requirements	EC, EP, ES
Sequencing & event timing requirements	EC, EP
Throughput & capacity requirements	EC, EP
Error Detection, Isolation, Recovery requirements	Ac, AM, Ds, Is, Sf
Quality Engineering requirements	ALL
Quality factors required	
Safety Requirements	AM, Sf, SA
Security and Privacy Requirements	
Access requirements	
to functions	Cm, Sf, SA
to data	Cm, Sf, SA
to code	Sf, SA
Legal requirements	Sf
Audit requirements	Vs
Other policy-based requirements	
Implementation Constraints	Ag, Ds, Ip
Site Adaptation	Ag, At, Gn
Design Goals	Cn, Cs, Gn, Sm
Human Factors Requirements	
User type definition	
level of computer sophistication	Op, Cc
technical competence required	Op, Cc
Physical constraints	
response time	Cm, Op
special physical limitations/requirements	Cm, Op
On-line help requirements	Op
Robustness requirements	AM, Gn, Sf, SA
Failure message & diagnostic requirements	AM, Cm, Cc, Gn, Is, Op
Input/Output convenience requirements	Cm, Cc, Is, Op
defaults	
formats	
Traceability to Parent's Design	Tc, Sm
Partitioning for Phased Delivery	DQ, Tc, Vs

Design Volume
- Architectural Design
  - Design Approach and Tradeoffs
  - Architectural Design Description
  - External Interface Design
  - Requirements Allocation and Traceability
  - Partitioning for Incremental Development
- Detailed Design
  - Detailed Design Approach and Tradeoffs
  - Detailed Design Description
  - External Interface Detailed Design
  - Coding and Implementation Notes
  - Firmware Support Manual

Version Description Volume
- Product Description
- Inventory and Product
  - Materials Released
  - Product Content

- Change Status
  - Installed Changes
  - Waivers
  - Possible Problems and Known Errors

User Documentation Volume
- User's Guide
  - Overview of Purpose and Function
  - Installation and Initialization
  - Startup and Termination
  - Functions and their Operation
  - Error and Warning Messages
  - Recovery Steps
- User's Training Materials

Maintenance Manual Volume
- Implementation Details
- Modification Aids
- Code Adaptation
- *Standards*

Abbreviations and Acronyms

Glossary

Notes

Appendices

Appendix C
DESIGN METHODOLOGIES and their INFORMATION PRODUCTS

This appendix lists information products generated by the more popular analysis & design methodologies of the day (compiled from [3], [9]). These products make up a portion of the contents of the Software Product Spec as listed in Appendix A and Appendix B. It is hoped to extend the tailoring method to recommend an appropriate set of design methodology information products based on the quality profile.

Functional Decomposition

Structured Design (SD) -- Constantine/Myers/Yourdon

This is the traditional data flow diagram methodology that has been in use since the early seventies. It's main products are a hierarchical set of data flow diagrams, process specifications and a data dictionary. State transition diagrams may also be used when deemed necessary by the analyst.

Real Time Structured Analysis & Design (RTSAD)

This methodology is similar to SD, but includes the analysis and design of control flow between processes. State transition diagrams, decision tables and process activation tables are used with more regularity.

Object Oriented Design (OOD)

OOD -- Booch

The objects defined in Booch's OOD have associated attributes and allowed operations. They use the concepts of visibility, class and inheritance, and they communicate with each other via message passing. One of Booch's goals in designing this methodology was to be compatible with the Ada language, and the objects map well to Ada constructs.

GOOD (General OOD) -- Seidewitz

The objects defined in this OOD have associated attributes only. They are tied to one another not by message passing, but by defined relationships. This is an attempt to model the real world more closely, and applies well to non-real time applications.

Other Methodologies

Jackson Structured Design (JSD) -- Jackson

This unique approach was an early contender on the requirements modeling scene, and is still going strong. As industry has developed the terms, we discover that JSD is a natural hybrid of Object Oriented and Functional Decomposition methodologies. JSD has its own set of information products which do not match 100% any of the traditional products in the map below, but I show what traditional products are most like those produced by JSD, rather than specifying and defining new product categories.

Ada-based Design Approach for Real Time Systems (ADARTS) -- Gomaa

This methodology is an Ada-based version of DARTS; it builds upon the SCR module structuring criteria, the Booch object structuring criteria, and the DARTS task structuring criteria to generate maintainable and reusable software components. It offers consideration of the concurrent nature of real-time systems. The analysis and design diagrams use the "Booch-gram" Ada notation.

Software Cost Reduction (SCR) -- Parnas

This real-time oriented methodology concentrates on the modules that will make up the software product, an information-hiding hierarchy into which they fall, and the interfaces which they use among themselves. Without trying, it is almost object oriented. The methodology offers strong support for software reuse.

Software Productivity Consortium Methodology (SPCM) -- Gomaa

This methodology is based on SCR. Its primary areas of focus are the inclusion of rapid prototyping techniques and the production of reusable software.

Information Products of the Methodologies

Product	*Methodologies which support generation of product*							
Context Diagram	SD	Rtsad						
Data Flow Diagrams	SD	Rtsad		GOOD	JSD	Adarts		
Control Flow Diagrams	SD	Rtsad						
Control Transformations (State Transitions)	SD	Rtsad	OOD	GOOD	JSD	Adarts	SCR	SPCM
Mini-Specs	SD	Rtsad						
Data Dictionary	SD	Rtsad			JSD			
Structure Charts	SD	Rtsad			JSD	Adarts		
Hardware Diagram			OOD					
Class Structure Diagram			OOD					
Architecture Diagram			OOD					
Ada Package Specs			OOD					
Object Diagram			OOD	GOOD	JSD			
Entity-Relation Diagrams		Rtsad		GOOD				
Process Definitions				GOOD				SPCM
Object Composition				GOOD				
Object Descriptions				GOOD				
Task Structure Specs						Adarts	SCR	SPCM
Module Guide						Adarts	SCR	SPCM
Module Interface Specs			OOD	GOOD		Adarts	SCR	SPCM
"Uses" Structure						Adarts	SCR	SPCM
Module Internal Design Spec							SCR	SPCM

Appendix D
OTHER SOFTWARE PROCESS MODELS

A sampling of Software Process Models other than the Waterfall Model are briefly described here. Recall that their associated information products are very similar to those described in Appendix A.

Spiral

A management oriented model. Activities and products are almost identical to those of the waterfall model, but are interspersed with regular prototyping and risk analyses efforts to guide the process.

Rapid Prototyping

This prototyping model covers the requirements definition phases of the waterfall or other similar model. It is generally recommended for never-before-attempted solutions, or when the user & developer deem areas of the problem concept to be technologically difficult.

A partial implementation of the system is constructed from *informal requirements*, usually of *poorly understood* areas. Users exercise of the prototype to better understand and define requirements. The prototype must then be discarded, and system design is begun from the requirements.

It is important to avoid temptations to keep and build upon the prototype, because the very nature of *rapid* prototyping causes generation of code that is inefficient, unsafe, unreliable, unmaintainable, etc. If, during development of the prototype, algorithms or designs are discovered that are particularly efficient, safe, reliable, maintainable, etc, they should be documented for consideration during the "real" design.

Evolutionary Prototyping

This prototyping model is also recommended for technologically difficult problems, but covers a larger area of the life cycle. It is hoped that the evolutionary prototyping efforts will help guide and speed the requirements definition, system design and implementation phases.

A partial implementation of the system is constructed from *partially known, well defined requirements*, usually of *well understood* areas. Users exercise the prototype to better understand and define remaining requirements. The prototype forms a set of baseline software which will be built upon to complete the deliverable versions. At this point, the model *may* transition to the Iterative Enhancement model.

Development of an evolutionary prototype begins with well defined requirements. It takes longer than rapid prototyping, because good software engineering practices must be used to develop code that will eventually be part of the working product.

Iterative Enhancement a.k.a. Incremental Development

This model is recommended for applications that have a basic, well understood core set of functions. The model is characterized by many releases of new versions which add new functionality. Many market-penetration schemes will use this model to get a product into the marketplace and generating revenue, to pay for later enhancements. A rather complete set of requirements is known up front, and the releases of new functions are planned in advance; of course, the model is adaptable to new requirements and relies on user feedback to improve the product.

Software Reuse

This model may be used to cover the design portion of the waterfall or other similar model. It's design paradigm relies mostly on the incorporation of previously proven designs and code into new software products.

Automated Software Synthesis

This is an advanced model that usually requires strict formulation of requirements using a regular grammar specification language. This model offers the direct (and hopefully, automatic) transformation of requirements and/or high level design into code, either algorithmically or using a knowledge based rule set. It is hoped to eliminate the middle portions of the documentation set, centering around the detailed design.

CASE tools currently exist that support this model to some degree. Typically, they will generate Ada package specs and the interface portions of package bodies from structure charts.

REFERENCES

[1] Abbot, R., *An Integrated Approach to Software Development*, John Wiley & Sons, NY 1986.

[2] Basili, V.; Rombach, H., "Tailoring the Software Process to Project Goals and Environments", *9th International Conference on Software Engineering*, IEEE Computer Society, Washington, DC 1987.

[3] Davis, A., "A Comparison of Techniques for the Specification of External System Behavior", *Communications of the ACM*, 31,9 (September 1988).

[4] Davis, A.; Bersoff, E.; Comer, E., "A Strategy for Comparing Alternative Software Development Life Cycle Models" *IEEE Transactions on Software Engineering*, 14,10 (October 1988).

[5] Deutsch, M.; Willis, R., *Software Quality Engineering: A Total Technical and Management Approach*, Prentice-Hall, Englewood Cliffs, NJ 1988.

[6] DOD-STD-2167A, *Military Standard: Defense System Software Development*, Department of Defense, Washington, DC, 1988.

[7] Fox, G., "Performance Engineering as a Part of the Development Life Cycle for Large-Scale Software Systems" *11th International Conference on Software Engineering*, IEEE Computer Society, Washington, DC 1989.

[8] Gilb, T., *Software Metrics*, Winthrop Publishers, Cambridge, 1977.

[9] Gomaa, H.; Kirby, J.; Weiss, D., "Comparison of Software Development Methodologies",*Presentation at Software Productivity Consortium Methodology Workshop*, March 1989.

[10] Humphrey, W., *Managing the Software Process*, Addison-Wesley, Reading, MA 1989.

[11] Humphrey, W., "Software Process Modeling: Principles of Entity Process Models" *9th International Conference on Software Engineering*, IEEE Computer Society, Washington, DC 1987.

[12] IEEE, *Software Engineering Standards*, IEEE Computer Society, Washington, DC 1987.

[13] Jackson, M., *System Development*, Prentice-Hall, Englewood Cliffs, NJ 1983.

[14] Krasner, H.; Pore, M., "A Software Process Management Approach to Quality and Productivity", Lockheed Software Technology Center, 1989.

[15] NASA, *Software Management and Assurance Program (SMAP) Information System Life Cycle and Documentation Standards Release 4.3*, NASA Office of Safety, Reliability, Maintainability and Quality Assurance, 1989.

[16] Poore, J., "Derivation of Local Software Quality Metrics (Software Quality Circles)" *Software Practice and Experience*, 18,11 (November 1988).

[17] Pressman, R., *Making Software Engineering Happen: A Guide for Instituting the Technology*, Prentice-Hall, Englewood Cliffs, NJ 1988.

[18] Pressman, R., *Software Engineering: A Practitioner's Approach*, McGraw-Hill, NY 1982.

[19] Rowen, R., "Software Project Management Under Incomplete and Ambiguous Specifications" *IEEE Transactions on Engineering Management*, 37,1 (February 1990).

[20] Tully, C., *Proceedings, 4th International Software Process Workshop*, ACM Press, NY 1989.

[21] Vincent, J.; Waters, A.; Sinclair, J., *Software Quality Assurance, Volume 1: Practice and Implementation*, Prentice-Hall, Englewood Cliffs, NJ 1988.