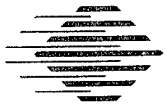
Handbook
CMU/SEI-97-HB-001



Carnegie-Mellon University
Software Engineering Institute

C4 Software Technology Reference Guide— A Prototype

Michael Bray
Kimberly Brune
David A. Fisher
John Foreman
Capt Mark Gerken
Jon Gross
Capt Gary Haines
Elizabeth Kean
Maj David Luginbuhl
William Mills
Robert Rosenstein
Darleen Sadoski
James Shimp
Edmond Van Doren
Cory Vondrak

January 10, 1997

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

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

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

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

Handbook

CMU/SEI-97-HB-001
January 1997

C4 Software Technology Reference Guide —A Prototype

Michael Bray, Lockheed Martin

Kimberly Brune, SEI

David A. Fisher, SEI

John Foreman, SEI

Capt Mark Gerken, USAF, Rome Laboratory

Jon Gross, SEI

Capt Gary Haines, USAF, AFMC SSSG

Elizabeth Kean, Rome Laboratory

Maj David Luginbuhl, USAF, Air Force Office of Scientific Research

William Mills, Lockheed Martin

Robert Rosenstein, SEI

Darleen Sadoski, GTE

James Shimp, E-System

Edmond Van Doren, Kaman Sciences

Cory Vondrak, TRW

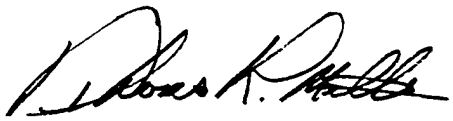
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

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



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

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

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

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

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

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center / 8725 John J. Kingman Road / Suite 0944 / Ft. Belvoir, VA 22060-6218. Phone: (703) 767-8222 *or toll-free in the U.S. — 1-800 225-3842).

Table of Contents

Foreword to the First Edition

This inaugural edition of the *C4 Software Technology Reference Guide*[1] marks the completion of an 11-month project undertaken by the Software Engineering Institute (SEI) and industry participants for the United States Air Force acquisition community. It includes the latest available information on approximately 60 software technologies. Even though this initial version tends to have a narrow customer focus, our hope is that this guide, with future editions, will become a common knowledge base and a valuable reference for the software engineering community. To our knowledge, no other document exists that provides this type of software technology information in one place.

Because software is a constantly-changing field, we are seeking sponsors, partners, and community participation to expand this document and publish descriptions of software technology on at least an annual basis. The long-term goal is a continuously growing and evolving Web-based reference guide of software technologies. This goal can only be reached with active participation and direct support from the broad software community. Please refer to Appendices A and B for details on how to contribute.

The document was developed as a cooperative effort among the SEI, six defense contractors, and three Air Force organizations. Project members were physically co-located during various intervals of the project, and also worked from geographically dispersed locations. Defense contractors and Air Force personnel authored technology descriptions and maintained/evolved the taxonomies and taxonomy-based directories. The SEI members established overall project direction, wrote various technology descriptions and sections, and integrated and edited the document extensively. We functioned as a cooperative team of individual contributors with a goal of technical accuracy and credibility. Team members were:

SEI

Kimberly Brune
David Fisher
John Foreman
Jon Gross
Robert Rosenstein

Defense Contractors

Michael Bray, Lockheed Martin
William Mills, Loral (Lockheed Martin)
Darleen Sadoski, GTE
James Shimp, E-Systems
Edmond Van Doren, Kaman Sciences

---

1. We have used other names for the document during earlier phases of the project. These include the *Software Technology Roadmap* and the *Structured Survey of Software Technology (SSST).*

Cory Vondrak, TRW

Air Force

Capt Mark Gerken, Rome Laboratory
Capt Gary Haines, AFMC SSSG
Elizabeth Kean, Rome Laboratory
Maj David Luginbuhl, Air Force Office of Scientific Research

In addition, other individuals contributed to the document by writing and/or reviewing technology descriptions. These contributions are acknowledged in the specific technology descriptions.

Development

The project entailed an intense data collection and writing effort by the authors. Because of the tight project schedule, the project team initially took an "opportunistic" approach to software technology topic coverage: The authors focused on software technologies in their areas of expertise, in areas where technical experts were accessible, and in areas where abundant supporting material existed. Topics selected were of significant interest to our targeted user community and/or were emerging technologies getting a lot of attention in the software community. Data collection approaches included the following:

- literature search and the World Wide Web
- previous studies, surveys, technical assessments, forecasts
- program briefings from DARPA, NSF, ATP, NIST, and many others
- interviews with researchers and technology developers in industry, government, and academia
- broad-based solicitation of written input

Each software technology description went through a rigorous review process which often included outside area experts. For the structure of the document, the *Physician's Desk Reference* (PDR), published by Medical Economics, provided us with a valuable model for overall organization. We referenced it often in the early days of the project. To ensure the development of a quality product, we also organized two formal outside reviews of the entire document. The first review team primarily focused on the technical validity and content of the document, while the second review team provided an executive management perspective on the utility of the document. Recommendations from these reviews were taken into consideration as we fine-tuned the document for this prototype publication.

We greatly appreciate the time and valuable input of the following review team members. Their insightful critique and recommendations significantly improved the quality of this initial publication.

Review Team #1 Participants

Dr. William L. Scherlis, Review Team Chair, Carnegie Mellon University
Ms. Christine Anderson, PL/VTQ
Dr. David J. Carney, Software Engineering Institute
Dr. John B. Goodenough, Software Engineering Institute
Lt. Col. Mike Goyden, SSSG/SMW
Col. Richard R. Gross, PhD, USAF, Defense Information Systems Agency
Lt. Col. J. Greg Hanson, PhD, HQ USAF/SCTS
Dr. Larry G. Jones, Software Engineering Institute
Mr. Mark H. Klein, Software Engineering Institute
Maj.George Newberry, SAF/AQRE
Ms. Linda M. Northrop, Software Engineering Institute
Lt. Col. Chris Ruminski, ESC/SRG
Dr. Howard E. Shrobe, Defense Advanced Research Projects Agency (DARPA)
Mr. John Willison, CECOM, Center for Software Engineering

Review Team #2 Participants

Dr. Harold W. Sorenson, Review Team Chair, The MITRE Corporation
Mr. Don Andres, TRW, Inc.
Mr. Archie Andrews, Software Engineering Institute
Mr. Thomas E. Bozek, Office of the Deputy Assistant Secretary of Defense (C3)
Mr. Thomas C. Brandt, Software Engineering Institute
Col. John Case, SSSG/SM
Mr. Clyde Chittister, Software Engineering Institute
Dr. Don Daniel, HQ AFMC/ST
Mr. Samuel A. DiNitto, Jr., Rome Laboratory
Dr. Larry E. Druffel, South Carolina Research Authority
Mr. John Gilligan, AF PEO/BA
Col. Richard R. Gross, PhD, USAF, Defense Information Systems Agency
Col. Robert A. Hobbs, USSPACECOM/J6N
Dr. Charles J. Holland, AFOSR/NM
Mr. John Kerschen, Lockheed Martin Command & Control Systems
Mr. Robert Knickerbocker, Lockheed Martin Command & Control Systems
Col. Robert Latiff, PhD, ESC/SR
Mr. Verlon Olson, Kaman Sciences Corporation
Mr. D. Michael Phillips, Software Engineering Institute
Mr. Randy Sablich, GTE Government Systems Corporation
Dr. William L. Scherlis, Carnegie Mellon University
Mr. Rick Sedlacek, E-Systems
Mr. Dennis Turner, US Army CECOM, Center for Software Engineering

It is our pleasure to acknowledge the following individuals who provided informative briefings and pragmatic reviews and insights:

Deane Bergstrom, Rome Laboratory
Dr. Jack Callahan, West Virginia University and NASA
Lt Col Thomas Croak, AFMC SSSG/SMX
Dr. Barbara B. Cuthill, NIST/ATP
Helen Gill, National Science Foundation
Dr. Richard Kieburtz, National Science Foundation
LTC(P) Mark Kindl, Army Research Labs
Jim Kirby, Naval Research Labs
Chuck Mertz, NASA
Dave Quinn, NSA
Dr. Howie Shrobe, DARPA
Bets Wald, Office of Naval Research

In addition, we would like to give special thanks to Karola Yourison, Sheila Rosenthal, and Terry Ireland of the SEI library and to Bernadette Chorle and Tamar Copeland for their support services. We would also like to thank the SEI Computing Facilities, Events, and Physical Facilities groups for their ongoing support of the project.

1 Introduction

1.1 Background

The Air Force acquisition community tasked the Software Engineering Institute (SEI) to create a reference document that would provide the Air Force with a better understanding of software technologies. This knowledge will allow the Air Force to systematically plan the research and development (R&D) and technology insertion required to meet current and future Air Force needs, from the upgrade and evolution of current systems to the development of new systems.

1.1.1 Scope

The initial release of the *Software Technology Reference Guide* is a prototype to provide initial capability, show the feasibility, and examine the usability of such a document. This prototype generally emphasizes software technology[1] of importance to the C4I (command, control, communications, computers, and intelligence) domain. This emphasis on C4I neither narrowed nor broadened the scope of the document; it did, however, provide guidance in seeking out requirements and technologies. It served as a reminder that this work is concerned with complex, large-scale, distributed, real-time, software-intensive, embedded systems in which reliability, availability, safety, security, performance, maintainability, and cost are major concerns.

We note, however, that these characteristics are not only applicable to military command and control systems, they apply as well to commercial systems, such as financial systems for electronic commerce. Also, for a variety of reasons, commercial software will play an increasingly important role in defense systems. Thus, it is important to understand trends and opportunities in software technology— including commercial software practice and commercially-available software components— that may affect C4I systems.

1.1.2 Vision

Our long-term goal is to create a continuously-updated, community "owned," widely-available reference document that will be used as a shared knowledge base. This shared knowledge base will assist in the tradeoff and selection of appropriate technologies to meet system goals, plan technology insertions, and possibly establish research agendas. While we use the term "document," we anticipate that this product will take many shapes, including a Web-based, paper-based, or CD-ROM based reference.

With the release of this document we are seeking comment and feedback from the software community. We will use this feedback as we plan an ongoing effort to expand and evolve this document to include additional software technology descriptions. Appendices A and B provide two vehicles by which readers can contribute to the further development of this effort.

---

1. This spectrum of technologies includes past, present, under-used, and emerging technologies.

1.1.3 Goal

The document is intended to be a guide to specific software technologies of interest to those building or maintaining systems, especially those in command, control, and/or communications applications. The document has many goals:

- to provide common ground by which contractors, commercial companies, researchers, government program offices, and software maintenance organizations may assess technologies
- to serve as *Cliff's Notes* for specific software technologies; to encapsulate a large amount of information so that the reader can rapidly read the basics and make a preliminary decision on whether further research is warranted
- to achieve objectivity, balance,[1] and a quantitative focus, bringing out both shortcomings as well as advantages, and provide insight into areas such as costs, risks, quality, ease of use, security, and alternatives
- to layer information so that readers can find subordinate technology descriptions (where they exist) to learn more about the topic(s) of specific interest, and to provide references to sources of more detailed technical information, to include usage and experience

1.1.4 Limitations/Caveats

While the document provides balanced coverage of a wide scope of technologies, there are certain constraints on the content of the document:

- *Coverage, accuracy and evolution.* Given the number of software technologies and the time available for this first release, this document covers a relatively small set of technologies. As such, there are many topics that have not been addressed; we plan to address these in subsequent versions. This document is, by nature, a snapshot that is based on what is known at the time of release. We have diligently worked to make the document as accurate as possible. A rating scheme describing the completeness of each technology description begins on pg. 49. Subsequent versions will include corrections and updates based on community feedback.
- *Not prescriptive.* This document is not prescriptive; it does not make recommendations, establish priorities, or dictate a specific path/approach.[2] The reader must make decisions about whether a technology is appropriate

---

1. As an example of balanced coverage, let's briefly look at information hiding of object-oriented inheritance, which reduces the amount of information a software developer must understand. Substantial evidence exists that such object-oriented technologies significantly increase productivity in the early stages of software development; however, there is also growing recognition that these same technologies may also encourage larger and less efficient implementations, extend development schedules beyond the "90% complete" point, undermine maintainability, and preclude error free implementations.

2. Similar to a roadmap for highways, the guide prescribes neither the destination nor the most appropriate route. Instead, it identifies a variety of alternative routes that are available, gives an indication of their condition, and describes where they may lead. Specific DoD applications must chart their own route through the technological advances.

for a specific engineering and programmatic context depending on the planned intended use, its maturity, other technologies that will be used, the specific time frame envisioned, and funding constraints.

For example, a specific technology may not be applicable to a particular program because the need is current and evaluations indicate that the technology is immature under certain circumstances. However, given a program that initiates in 3-5 years, the same technology may be an appropriate choice assuming that the areas of immaturity will be corrected by then (and, if necessary, directed action to ensure the maturation or to remedy deficiencies).

- *Not a product reference.* This document is not a survey or catalog of products. There are many reasons for this, including the rapid proliferation of products, the need to continually assess product capabilities, questions of perceived endorsement, and the fact that products are almost always a collection of technologies. It is up to the reader to decide which products are appropriate for their context. DataPro and Auerbach would likely be better sources of product-specific information.
- *Not an endorsement.* Inclusion or exclusion of a topic in this document does not constitute an endorsement of any type, or selection as any sort of "best technical practice." Judgements such as these must be made by the *readers* based on their contexts; our goal is to provide the balanced information to enable those judgements.
- *Not a market forecasting tool.* While the technology descriptions may project the effect of a technology and discuss trends, more complete technology market analysis and forecast reports are produced by organizations such as The Yankee Group, Gartner Group, and IDC.
- *Not a focused analysis of specific technical areas.* Various sources such as Ovum, Ltd. and The Standish Group offer reports on a subscription or one-time basis on topics such as workflow, open systems, and software project failure analyses, and may also produce specialized analyses and reporting on a consulting basis.

1.1.5 Target Audiences

We envisioned that this document would be relevant to many audiences. The audiences and a description of how each audience can use this document are shown in the table below.

User	Job Roles/Tasks	Document Capabilities/Value
PEO/Executive Pentagon Action Officer	Acquisition oversight, funding advocacy Motivate introduction of new/commercial technologies Policy issues	Overview/introductory info Baseline reference document "Cliff Notes" approach— provides high-level, 4-6 page quick study Tradeoff information
System Program Manager (SPM) and Technical Staff (Includes FFRDCs (MITRE, etc.) and may include government laboratories)	Writes Request for Proposal (RFP) or some form of solicitation based on user requirements Reviews proposals and selects developers Manages development and/or maintenance work	All of previous category, plus: Taxonomies to aid in identifying alternatives Back pointers to high-level, related technologies Criteria and guidance for decision-making Tech transfer/insertion guidelines Selected high-value references to more technical information, to include usage and experience data Generally the sort of analysis and survey information that would not be accomplished under normal project circumstances
Developer (to include research and development (R&D) activity)	Performs advanced development, prototyping, and technology investigation focused on risk reduction and securing competitive advantage Concerned about transition and insertion issues Writes a proposal in response to solicitations Performs engineering development and provides initial operational system	Same as previous category.
Maintainer	Maintains operational system until the end of the life cycle Responds to user requirements for corrections or enhancements Concerned about inserting new technologies and migrating to different approaches	Same as previous category.
User	Communicates operational needs End customer for operational system	Communicates alternatives and risks, and provides perspective of what technology can (reasonably) provide

1.2 Using the Document

While some readers may elect to read the document from cover to cover, the document provides several methods by which readers can locate technology descriptions in a more direct manner. These methods include the following:

Alphabetical listing. On page 59, the technology descriptions are listed in alphabetical order by title. If the reader has a specific software technology in mind, this method may be useful because the reader only has to perform an alphabetical search. In addition, readers may want to pay close attention to descriptions that have "An Overview" in their title. These descriptions provide an excellent starting point and set the context for a particular group of technologies.

Taxonomies and taxonomy-based directories. The taxonomies and taxonomy-based directories provide two valuable benefits for the reader:

1. The taxonomy-based directories group technology descriptions according to the taxonomy categories into which they have been categorized.
2. The taxonomies identify the relationships between taxonomy categories; this provides the reader with suggestions for other categories to search.

Details on how to use these features can be found in Section 2.

Keyword index. The Keyword Index is generally structured as in any typical document; a few nuances of our index are explained below:

- If the keyword happens to be the name of a technology, the index will guide the reader directly to the description for that technology by identifying the page number in bold type. For example, the index entry for "middleware" is

  middleware 79, 247, **251**, 291, 325, 373

  The technology description on middleware can be found on page 251.

- If the keyword is a category in one of the taxonomies, the index will direct the reader to the taxonomies in Chapter 2 by following the keyword entry with a taxonomy index label in italics. For example, the keyword entries for "reengineering" and "reliability" would show: reengineering (*AP.1.9.5*) and reliability (*QM.2.1.2*).

Cross-references. Cross-references are page references within a technology description to other technology descriptions.

Scenarios. Appendix C contains questions that typical C4 organizations might ask. Scenarios demonstrate how to use the document to answer or address a particular question.

2 Taxonomies

2.1 Overview and Purpose

Some readers may not desire to read the guide from cover to cover or may not have a specific technology in mind when they open the document. Instead a reader might be concerned about or interested in a particular software quality measure, a phase of the development process, or an operational function.

With this in mind, we created two taxonomies that serve as directories into the technology descriptions. This method is an effective way to lead readers to a set of possible technologies that address their software problem area. Each software technology description has been categorized into the following two taxonomies:

- **Application.** This taxonomy categorizes technologies by how they might be used in operational systems. A technology can fall into one of two major categories. It can be used to support an operational system or it can be used in an operational system.

- **Quality Measures.** This taxonomy categorizes technologies by the software quality characteristics or attributes that they influence, such as maintainability, expendability, reliability, trustworthiness, robustness, and cost of ownership.

The taxonomies serve other purposes as well. A taxonomy implies a hierarchical relationship of terms which are used for classifying items in a particular domain. It is this hierarchical relationship that we wanted to capture for the reader with the hope that each taxonomy would provide stand-alone utility. Additionally, this relationship of terms gives the reader an idea of alternative categories in which to look for technology descriptions.

Each taxonomy section contains the following:

- a graphical representation of the taxonomy
- a textual representation of the taxonomy
- a taxonomy-based directory with all technology descriptions appropriately categorized

Definitions for most of the taxonomy categories can be found in the Glossary, pg. 393.

2.1.1 General Taxonomy Structure

Both taxonomies are structured in a similar manner. Each term or category in a taxonomy has an index number. For the Application taxonomy, the index numbers begin with AP; for the Quality Measures taxonomy, the index numbers begin with QM. As mentioned before, a taxonomy is a hierarchical relationship. A category can be broken down into one or more subcategories with the subcategories beginning a new level in the hierarchy. Subcategories are indexed starting with the number 1. For example, index numbers that are subcategories to the

first, or root level (AP or QM) would look like AP.2, QM.1, or QM.3. Subcategories to AP.2, QM.1, or QM.3 would have index numbers like AP.2.4, QM.1.1, or QM.3.2, respectively; subcategories to these would have index numbers like AP.2.4.3, QM.1.1.2, or QM.3.2.1, respectively, and so on.

Some categories have hyphenated subcategories. These subcategories are terms that we feel are worth noting and help further define what type of technology descriptions the reader may find under the parent category. However, they are not sufficiently different from their parent category or in some cases from each other to warrant an index number.

Technology descriptions can be classified into more than one category, and these categories are usually three to four levels deep in the taxonomy.

The graphical and textual representations of each taxonomy do not have technology descriptions categorized into them. These representations are an aid to readers so that they can easily see the relationships between the different taxonomy terms. The index numbers do appear in these representations.

2.1.2 Using the Taxonomies

The taxonomies will most likely be referenced after the reader has first visited the Keyword Index. The Keyword Index will identify the index number for a term if it is a category within one of the taxonomies. This index number will help the reader find technology descriptions using the taxonomy-based directory. For example, if a reader is concerned about testing, the reader would first go to the Keyword Index. In the Keyword Index under "testing", the reader will find an index number of "AP.1.5.3". This tells the reader that testing is a category in the Application taxonomy and that the index number for testing is AP.1.5.3. In addition, the reader will also notice that under the "testing" entry are types of testing such as "interface," "operational," and "unit." These testing types also have index numbers: AP.1.5.3.3, AP.1.4.3.4, and AP.1.8.2.1, respectively. Now, the reader must to make a decision. The reader decides that unit testing is really the problem. The reader then turns to the Application taxonomy-based directory and looks for AP.1.4.3.4. Once found, the reader will see a list of technology descriptions that relate to unit testing.

When readers find a term within one of the taxonomies that leads them to a list of technology descriptions, they may want to examine the graphical or textual representations of the taxonomies as well. By examining these, readers can identify other possible categories to look under that are related to their original term. For example, if a reader is concerned about reliability, the reader would look at one of the Quality Measures representations and notice that "correctness" and "completeness" are closely related to reliability. The reader could then look for technology descriptions under those categories. This method may give the reader a more complete solution set for their particular problem context.

2.2 Application Taxonomy

2.2.1 Introduction

Readers will use the application taxonomy if they are looking for software technologies that address a particular use, such as design or testing. The technology descriptions have been classified into this taxonomy according to how they are used in systems. Specifically, the application taxonomy divides software technologies into two major categories:

1. Used to support operational systems
2. Used in operational systems

Under the category "Used to Support Operational Systems" (AP.1), by referencing ANSI/IEEE Std 1002-1987 [IEEE 1002], we provide the standard life cycle phases plus two major activities that cross all of the phases. IEEE Std 1074-1991 [IEEE 1074] helped provide a breakdown of the activities that occur in each life cycle phase. Support in this context means any technology used to develop and maintain an operational system within the life-cycle framework.

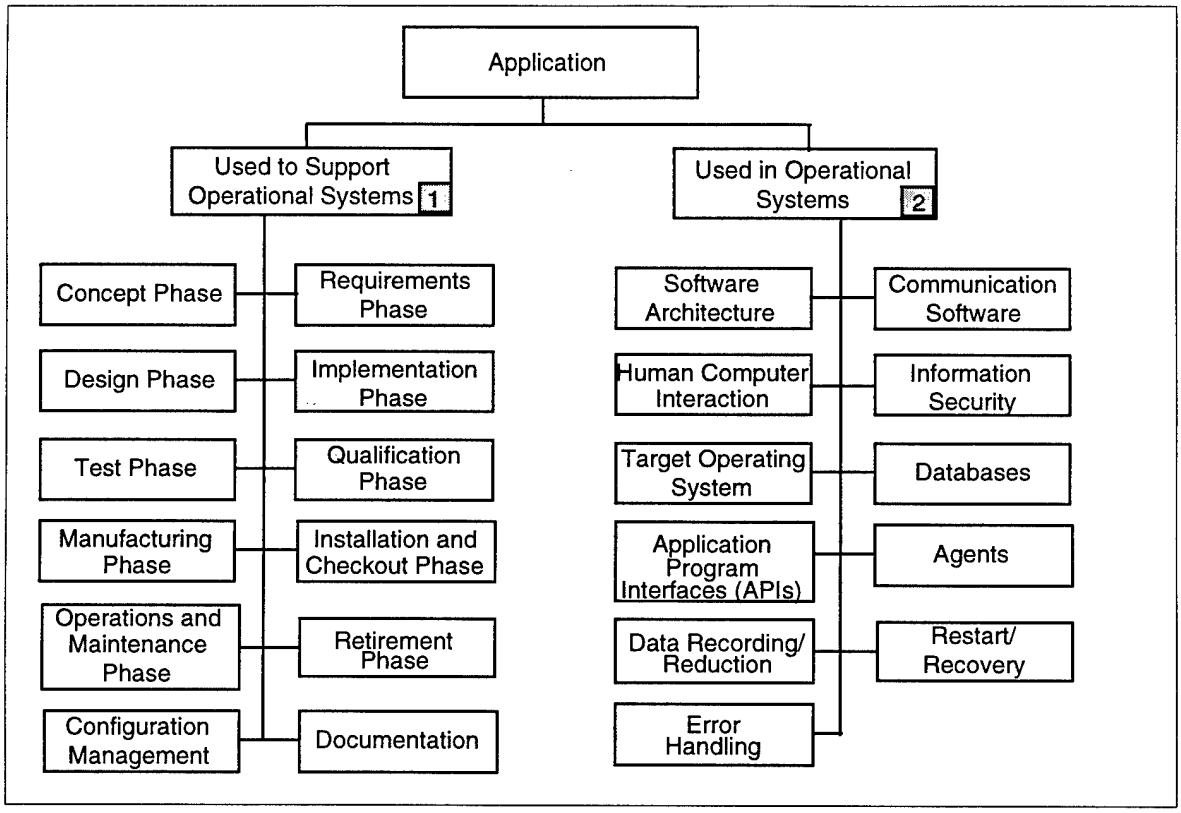
The category "Used in Operational Systems" (AP.2) simply provides a breakdown of categories of technologies that are used and operate in operational systems.

Note: Within the technology descriptions, some software technologies that are mentioned or referenced do not yet have corresponding descriptions. However, we still indexed these into the Application Taxonomy. When these descriptions are written and more information is gathered, the categories into which these technologies are indexed may change. Thus technologies may appear in this taxonomy without corresponding page references.

2.2.2 Graphical Representation

The following explains how to approach the graphical representations:

- There is always a two-level deep view from the root figure.
- Due to the structure of this taxonomy, it may take more than one figure to provide a complete two-level deep view.
- If further expansion of the taxonomy is needed (i.e., there is more detail at subordinate levels), the first level is marked with a number in a shaded box located in the lower, right-hand corner. That level is then further expanded (and rotated 90 degrees) in Figure X where X corresponds to the number that the level is marked with.



Root Figure: Application

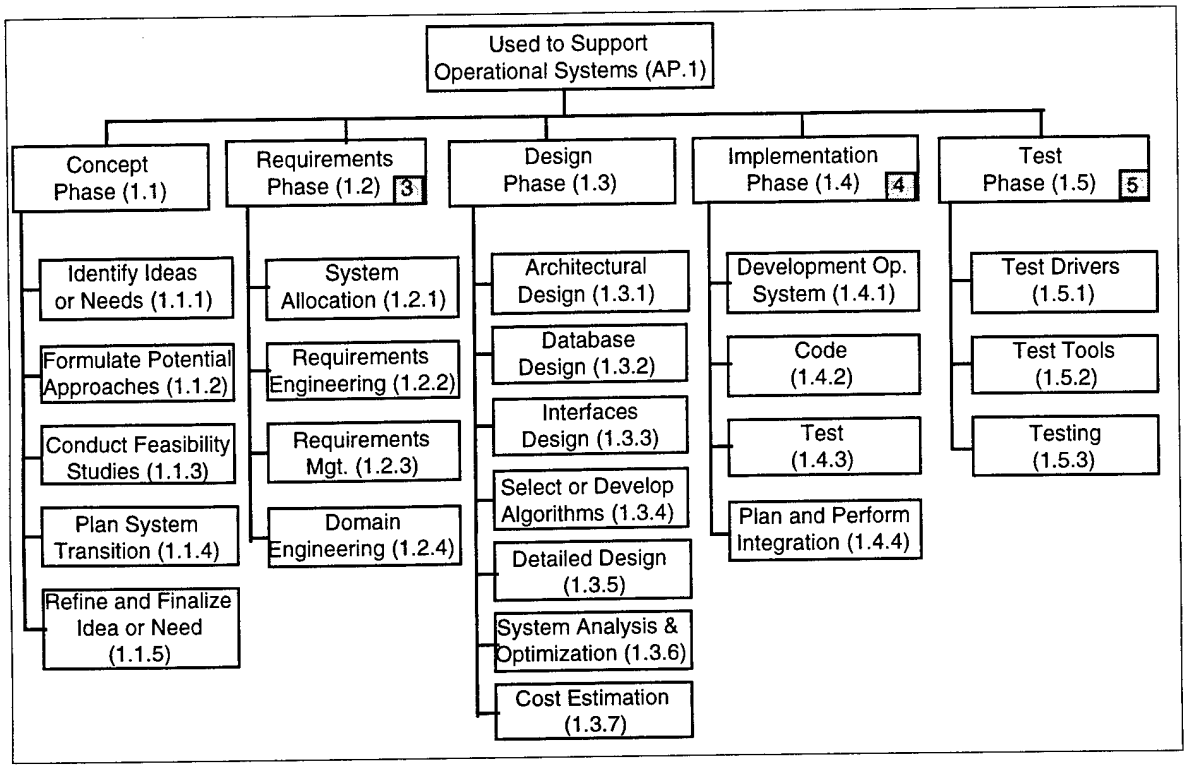


Figure 1a: Used to Support Operational Systems

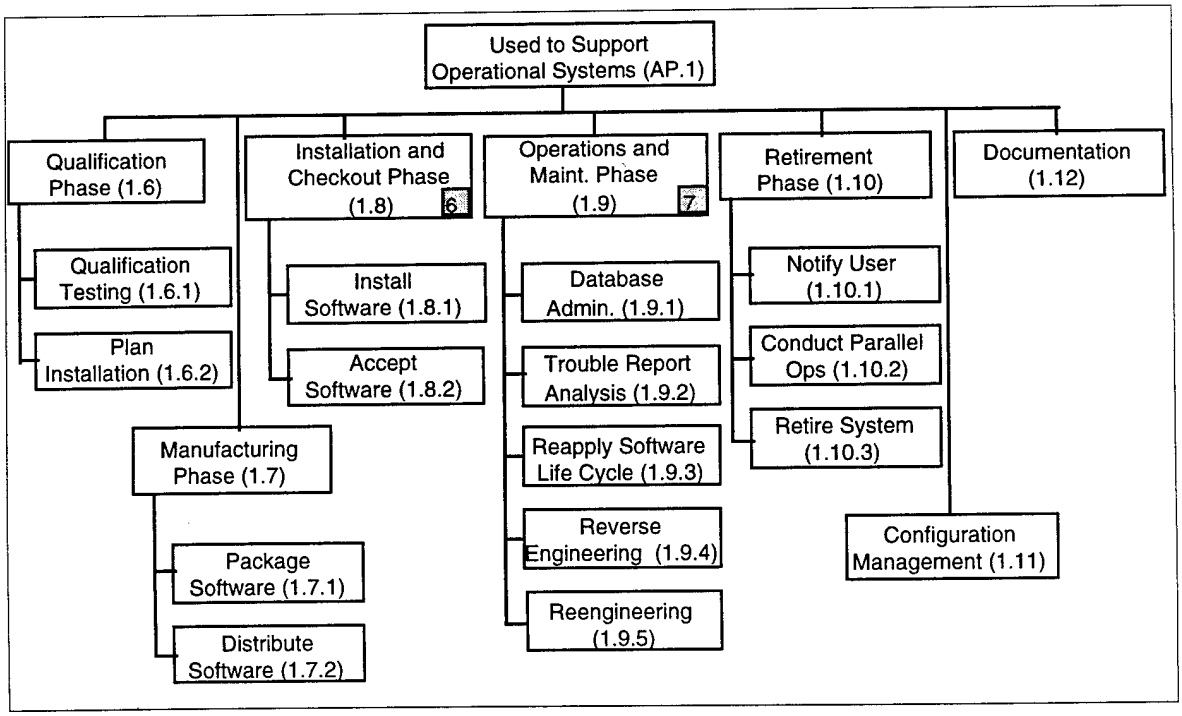


Figure 1b: Used to Support Operational Systems

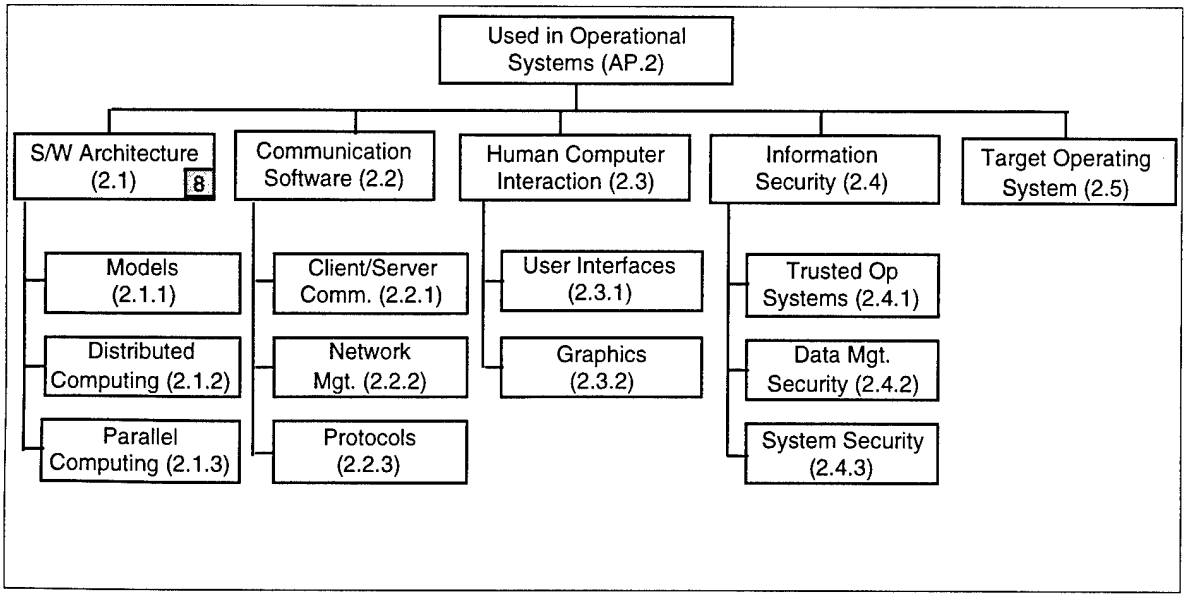


Figure 2a: Used in Operational Systems

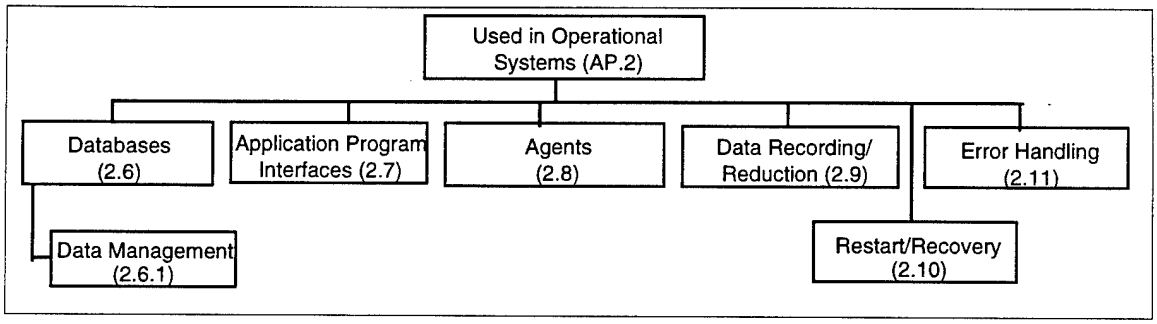


Figure 2b: Used in Operational Systems

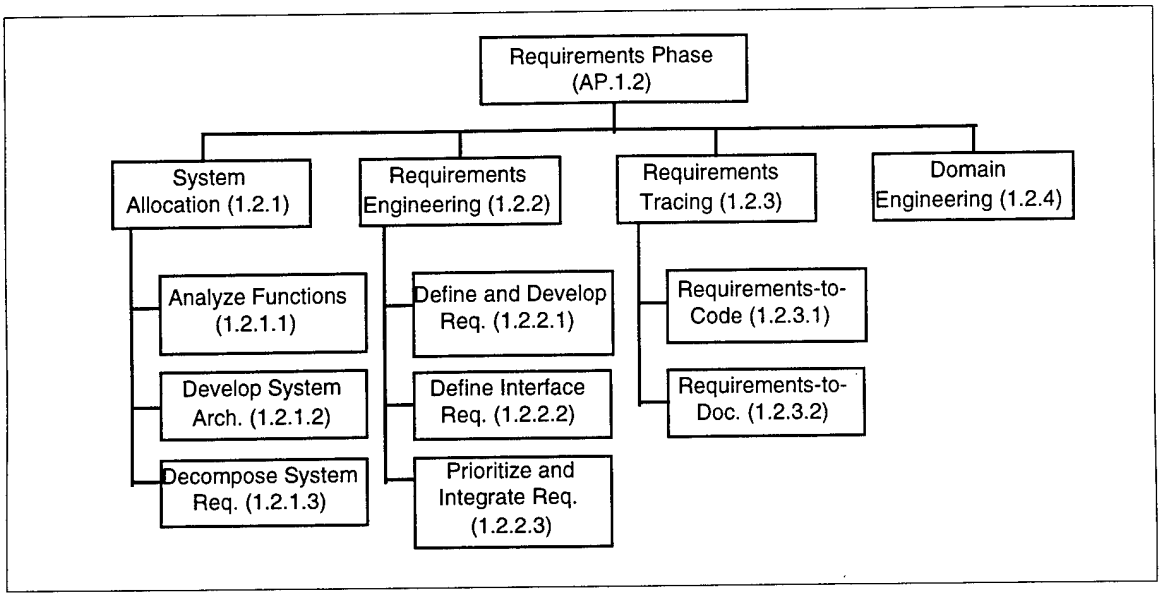


Figure 3: Requirements Phase

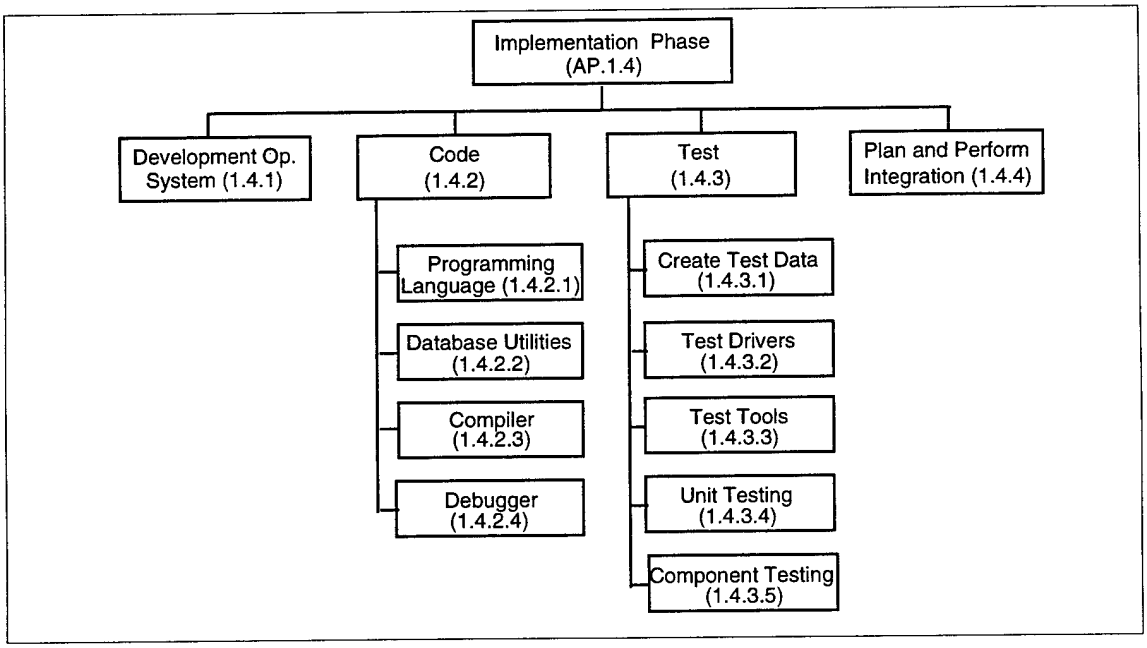


Figure 4: Implementation Phase

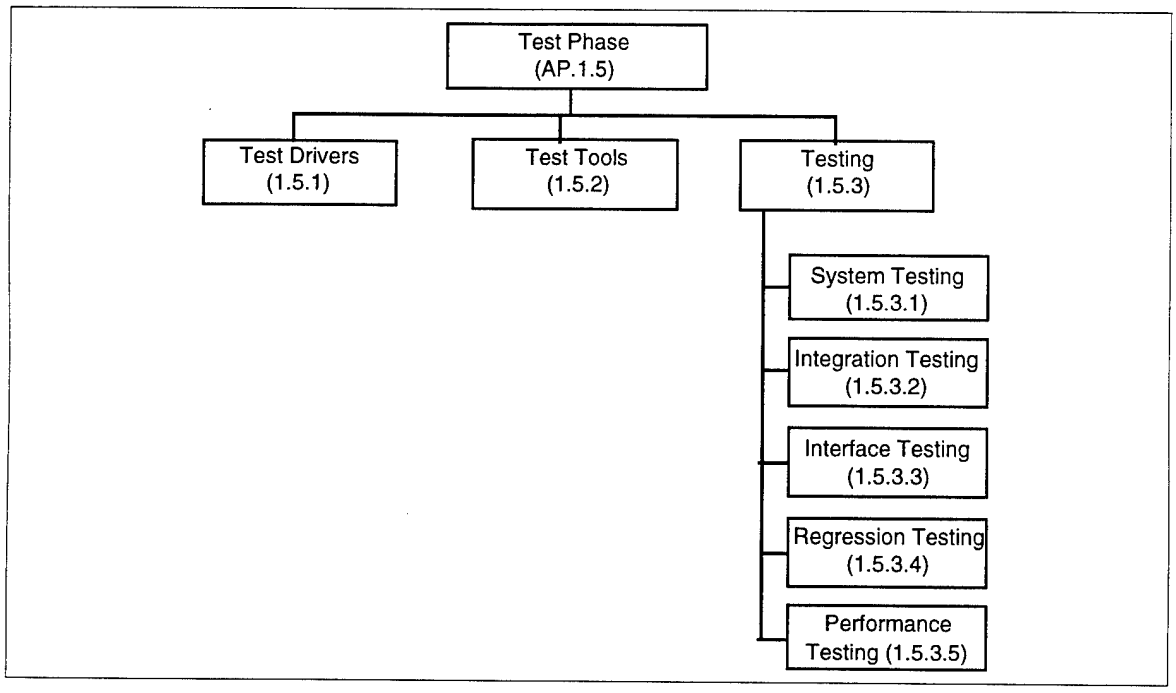


Figure 5: Test Phase

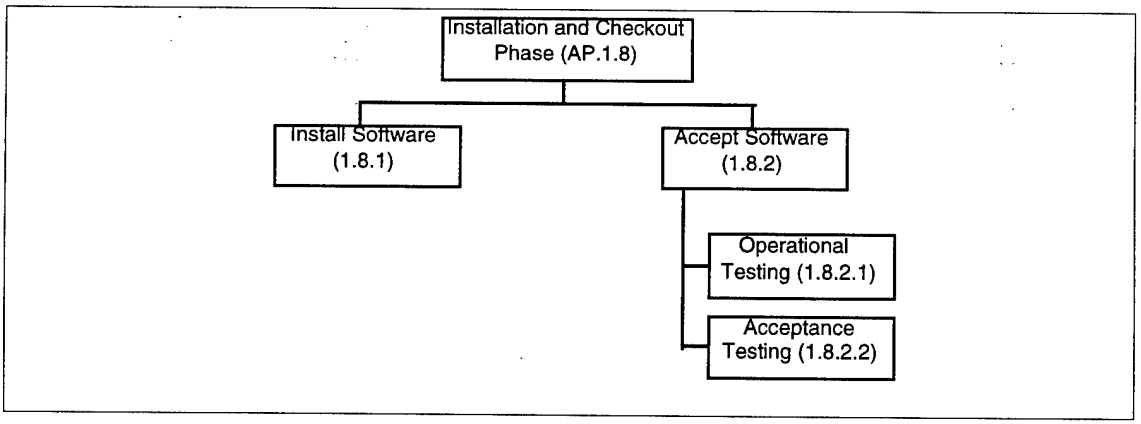


Figure 6: Installation and Checkout Phase

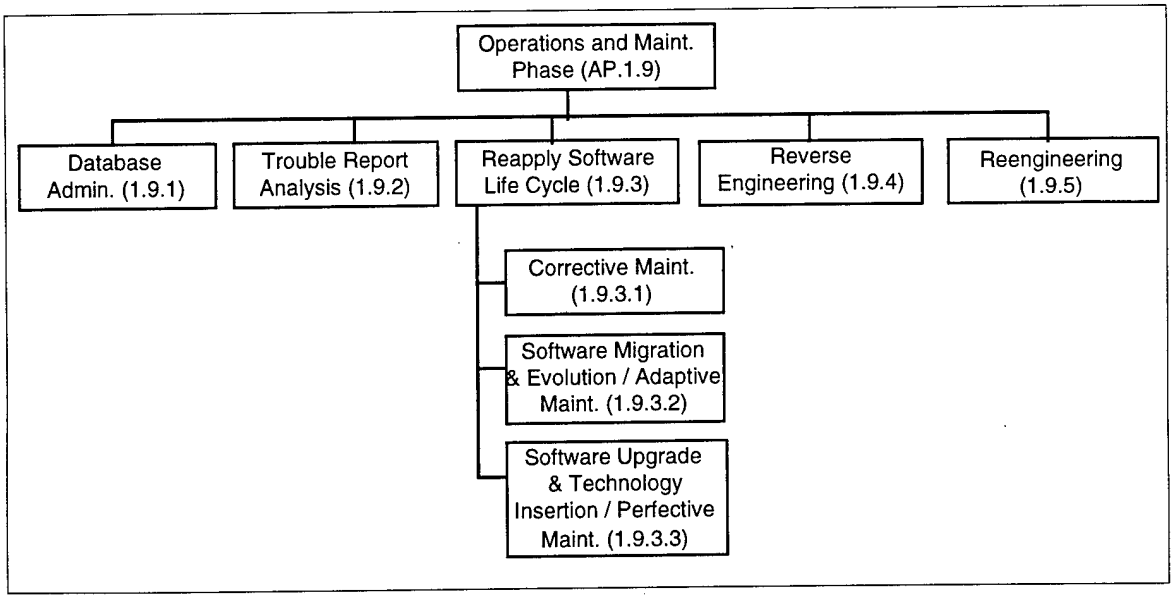


Figure 7: Operations and Maintenance Phase

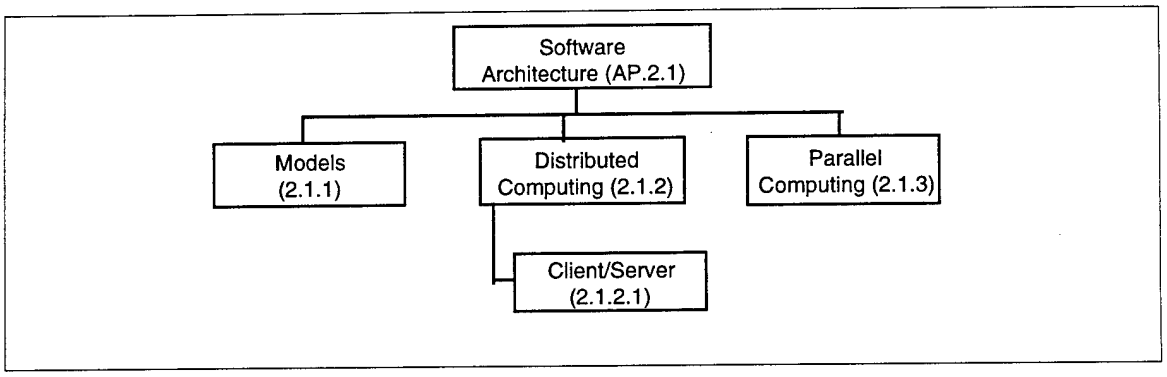


Figure 8: Software Architecture Phase

2.2.3 Textual Representation

AP. Application

1. Used to Support Operational Systems
    1. Concept Phase
        1. Identify Ideas or Needs
        2. Formulate Potential Approaches
        3. Conduct Feasibility Studies
        4. Plan System Transition
        5. Refine and Finalize Idea or Need
            - Collect Pertinent Documentation,Regulations, Procedures, and Policies
    2. Requirements Phase
        1. System Allocation
            1. Analyze Functions
            2. Develop System Architecture
            3. Decompose System Requirements
        2. Requirements Engineering
            1. Define and Develop Requirements
                - Elicitation Techniques
                - Specification Techniques
                - Modeling
                - Prototyping
            2. Define Interface Requirements
            3. Prioritize and Integrate Requirements
        3. Requirements Tracing
            1. Requirements-to-code
            2. Requirements-to-documentation
        4. Domain Engineering
    3. Design Phase
        1. Architectural Design
            - Hardware-Software Co-Design
        2. Database Design
            - Conceptual
            - Logical
            - Physical
        3. Interfaces Design
        4. Select or Develop Algorithms
        5. Detailed Design
            - Design Notations
            - Design Techniques
        6. System Analysis and Optimization
        7. Cost Estimation
    4. Implementation Phase
        1. Development Operating System
        2. Code
            1. Programming Language

2. Database Utilities
3. Compiler
4. Debugger

3. Test
    1. Create Test Data
    2. Test Drivers
    3. Test Tools
    4. Unit Testing
        - Code analyzers
        - Data analyzers
        - Black-box/Functional Testing
        - White-box/Structural Testing
    5. Component Testing
4. Plan and Perform Integration

5. Test Phase
    1. Test Drivers
    2. Test Tools
    3. Testing
        1. System Testing
        2. Integration Testing
        3. Interface Testing
        4. Regression Testing
        5. Performance Testing
            - Statistical Testing

6. Qualification Phase
    1. Qualification Testing
    2. Plan Installation

7. Manufacturing Phase
    1. Package Software
    2. Distribute Software

8. Installation and Checkout Phase
    1. Install Software
    2. Accept Software
        1. Operational Testing
        2. Acceptance Testing

9. Operations and Maintenance Phase
    1. Database Administration
    2. Trouble Report Analysis
    3. Reapply Software Life Cycle
        1. Corrective Maintenance
        2. Software Migration and Evolution / Adaptive Maintenance
        3. Software Upgrade and Technology Insertion / Perfective Maintenance
    4. Reverse Engineering
    5. Reengineering

10. Retirement Phase

1. Notify User
2. Conduct Parallel Operations
3. Retire System

11. Configuration Management
12. Documentation

2. Used in Operational Systems
    1. Software Architecture
        1. Models
        2. Distributed Computing
            1. Client/Server
        3. Parallel Computing
    2. Communication Software
        1. Client/Server Communication
        2. Network Management
        3. Protocols
    3. Human Computer Interaction
        1. User Interfaces
        2. Graphics
    4. Information Security
        1. Trusted Operating Systems
        2. Data Management Security
        3. System Security
    5. Target Operating Systems
    6. Databases
        1. Data Management
    7. Application Program Interfaces (APIs)
    8. Agents
    9. Data Recording/Reduction
    10. Restart/Recovery
    11. Error Handling

2.2.4 Taxonomy-Based Directory to Technology Descriptions

AP. Application

AP.1 Used to Support Operational Systems

AP.1.1 Concept Phase

AP.1.1.1 Identify Ideas or Needs

AP.1.1.2 Formulate Potential Approaches

AP.1.1.3 Conduct Feasibility Studies

AP.1.1.4 Plan System Transition

AP.1.1.5 Refine and Finalize Idea or Need (Collect Pertinent Documentation, Regulations, Procedures, and Policies)

AP.1.2 Requirements Phase

AP.1.2.1 System Allocation

Component-Based Software Development/ COTS Integration 119

AP.1.2.1.1 Analyze Functions

Essential Systems Analysis

Functional Decomposition

Object-Oriented Analysis 275

Structured Analysis and Design

AP.1.2.1.2 Develop System Architecture

AP.1.2.1.3 Decompose System Requirements

AP.1.2.2 Requirements Engineering

AP.1.2.2.1 Define and Develop Requirements (Elicitation Techniques, Specification Techniques, Modeling, Prototyping)

Algebraic Specification Techniques

Box Structure Method

Entity-Relationship Modeling

Essential Systems Analysis

Formal Specification

Functional Decomposition

Model Checking

Object-Oriented Analysis 275

Specification Construction Techniques

Structured Analysis and Design

AP.1.2.2.2 Define Interface Requirements

AP.1.2.2.3 Prioritize and Integrate Requirements

AP.1.2.3 Requirements Tracing

Argument-Based Design Rationale Capture Methods for Requirements Tracing 91

Feature-Based Design Rationale Capture Method for Requirements Tracing. 181

Maintenance of Operational Systems— an Overview 237

Representation and Maintenance of Process Knowledge Method Requirements Tracing 327

AP.1.2.3.1 Requirements-to-code

AP.1.2.3.2 Requirements-to-documentation

AP.1.4.2.3 Compiler

AP.1.4.2.4 Debugger

AP.1.4.3 Test

AP.1.4.3.1 Create Test Data

AP.1.4.3.2 Test Drivers

AP.1.4.3.3 Test Tools

AP.1.4.3.4 Unit Testing (Code analyzers, Data analyzers, Black-box/Functional Testing, White-box/Structural Testing)

2.3 Quality Measures Taxonomy

2.3.1 Introduction

Readers will use the quality measures taxonomy if they are looking for software technologies that affect particular quality measures or attributes of a software component or system. The technology descriptions have been categorized into this taxonomy by the particular quality measure(s) that they directly influence. Software quality can be defined as the degree to which software possesses a desired combination of attributes (e.g., reliability, interoperability) [IEEE 90]. Software technologies are typically developed to affect certain quality measures.

We developed a reasonably exhaustive and non-overlapping set of measures by which the quality of software is judged. With the help of work done by Boehm, Barbacci, Deutsch and Willis, and Evans and Marciniak, we established a hierarchical relationship among our list of quality measures to create the taxonomy [Boehm 78, Barbacci 95, Deutsch 88, Evans 87]. The following table explains the categories of quality measures and the areas they address:

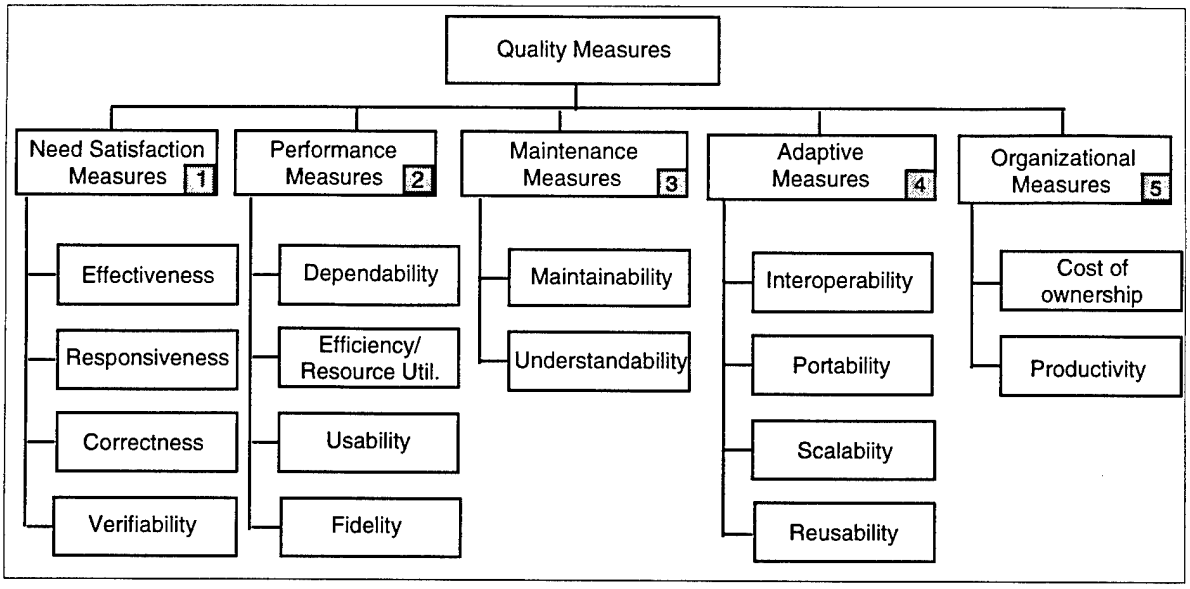
Quality Measure	**Area Addressed**
Need Satisfaction (QM.1)	How well does the system meet the user's needs and requirements?
Performance (QM.2)	How well does the system function?
Maintenance (QM.3)	How easily can the system be repaired or changed?
Adaptive (QM.4)	How easily can the system evolve or migrate?
Organizational (QM.5)	none specifically, usually indirect

Categories 1 - 4 are all considered to be direct measures, i.e., quality attributes that can be directly impacted by software technologies. The measures listed in category 5 are measures that generally can not be affected directly by software technologies, but have an indirect relationship. Many factors influence these measures, such as management, politics, bureaucracy, employee skill-level, and work environment. For example, software alone can not improve productivity. A software technology that improves a direct measure such as understandability may indirectly improve productivity. Therefore, most technology descriptions will *not* be categorized into category 5. An example of a technology the reader may find in this category is a technology that was specifically developed to measure or estimate costs of productivity associated with software.

2.3.2 Graphical Representation

The following explains how to approach the graphical representations:

- There is always a two-level deep view from the root figure.
- If further expansion of the taxonomy is needed (i.e., there is more detail at subordinate levels), the first level is marked with a number in a shaded box located in the lower, right-hand corner. That level is then further expanded (and rotated 90 degrees) in Figure X where X corresponds to the number that the level is marked with.



Root Figure: Quality Measures

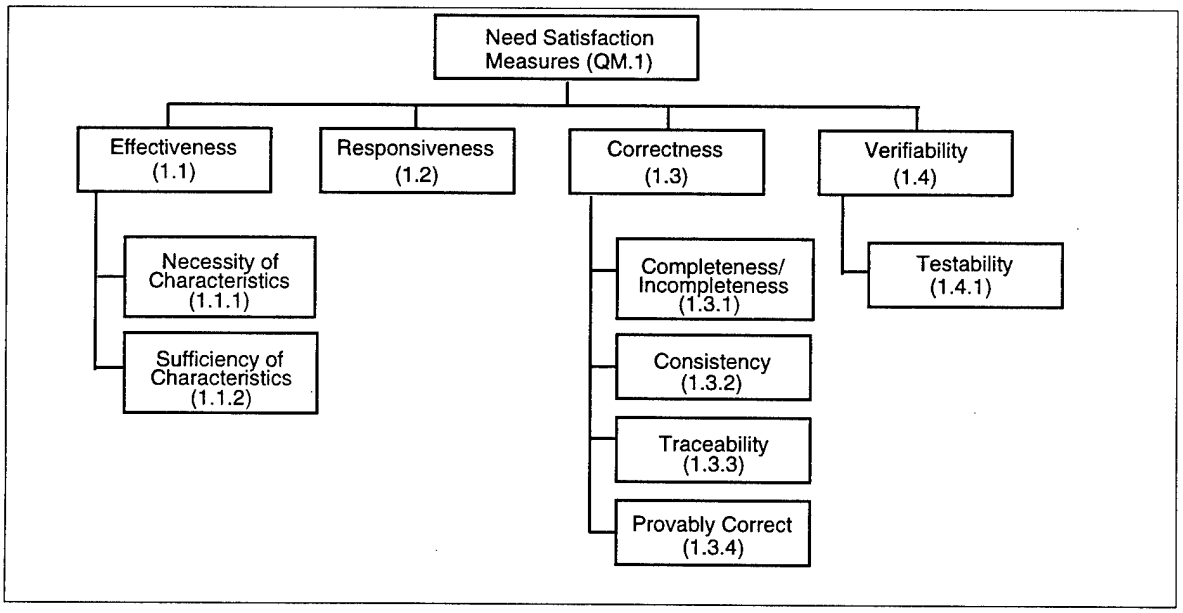


Figure 1: Needs Satisfaction Measures

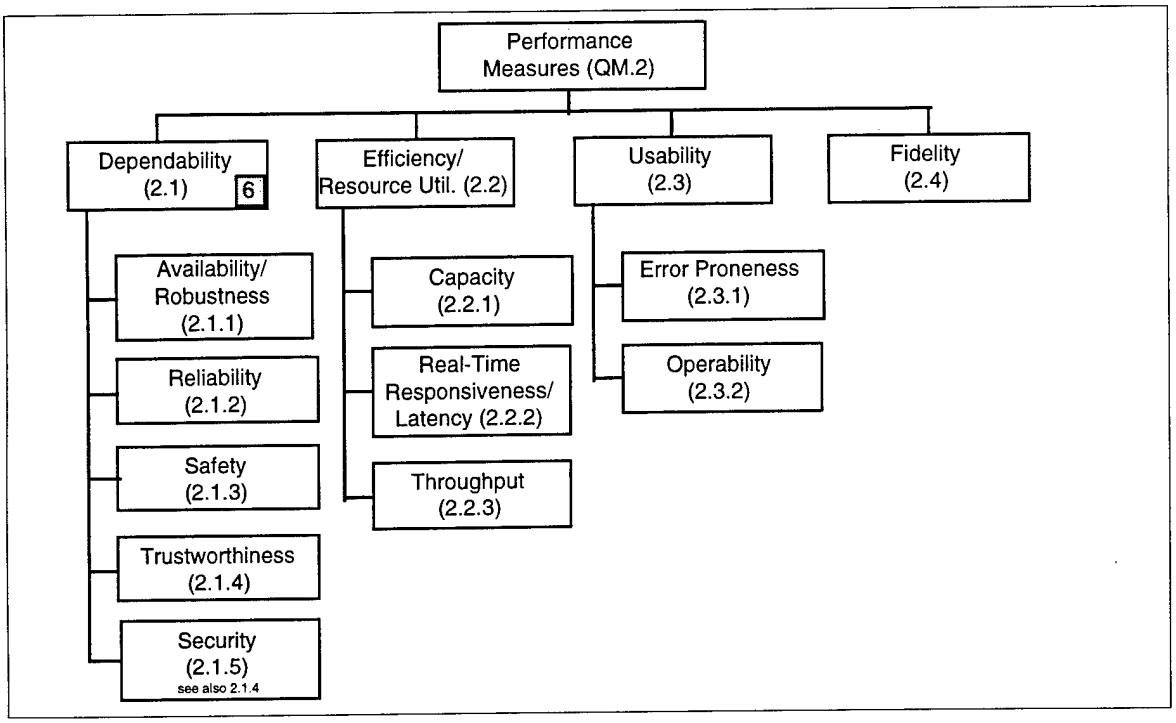


Figure 2: Performance Measures

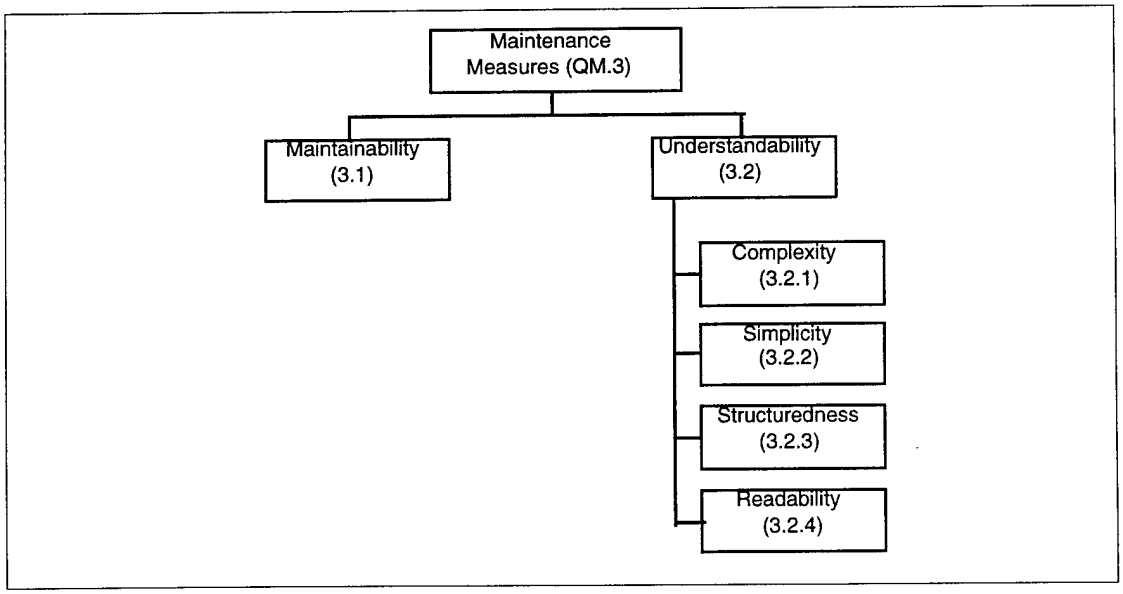


Figure 3: Maintenance Measures

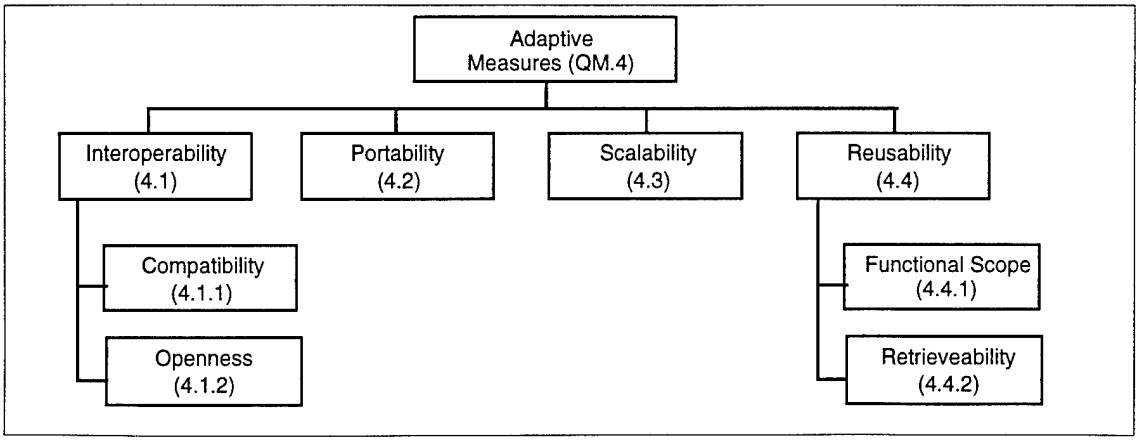


Figure 4: Adaptive Measures

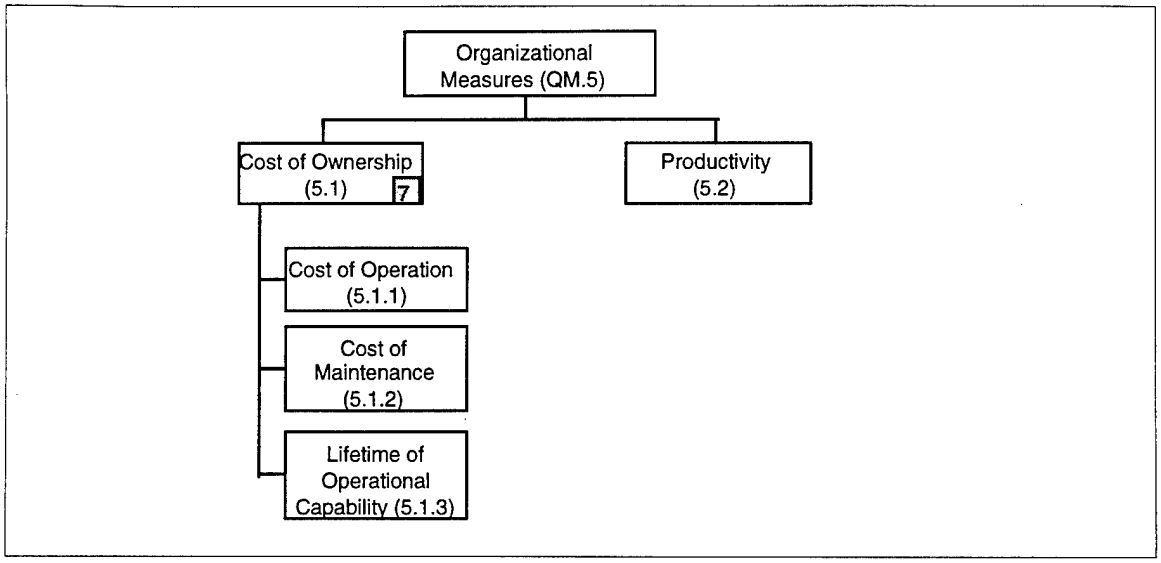


Figure 5: Organizational Measures

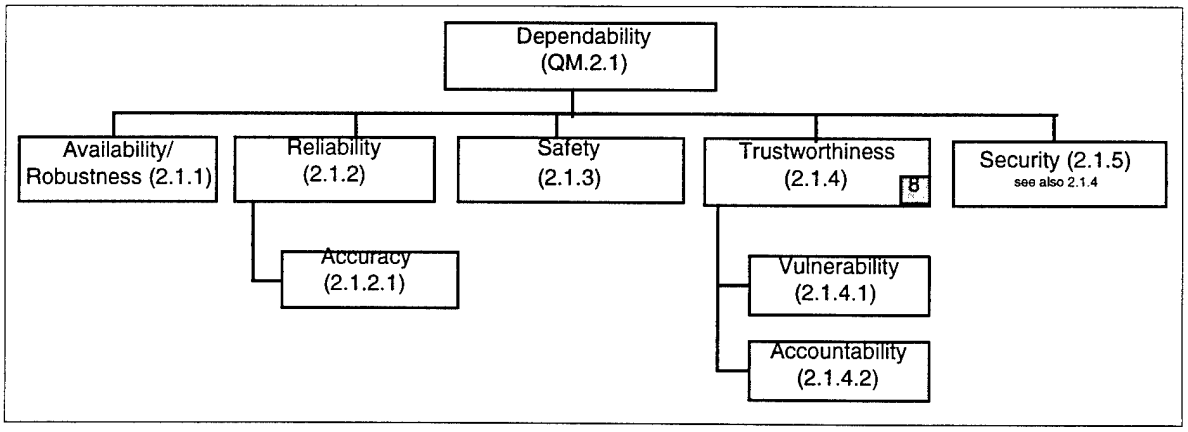


Figure 6: Dependability

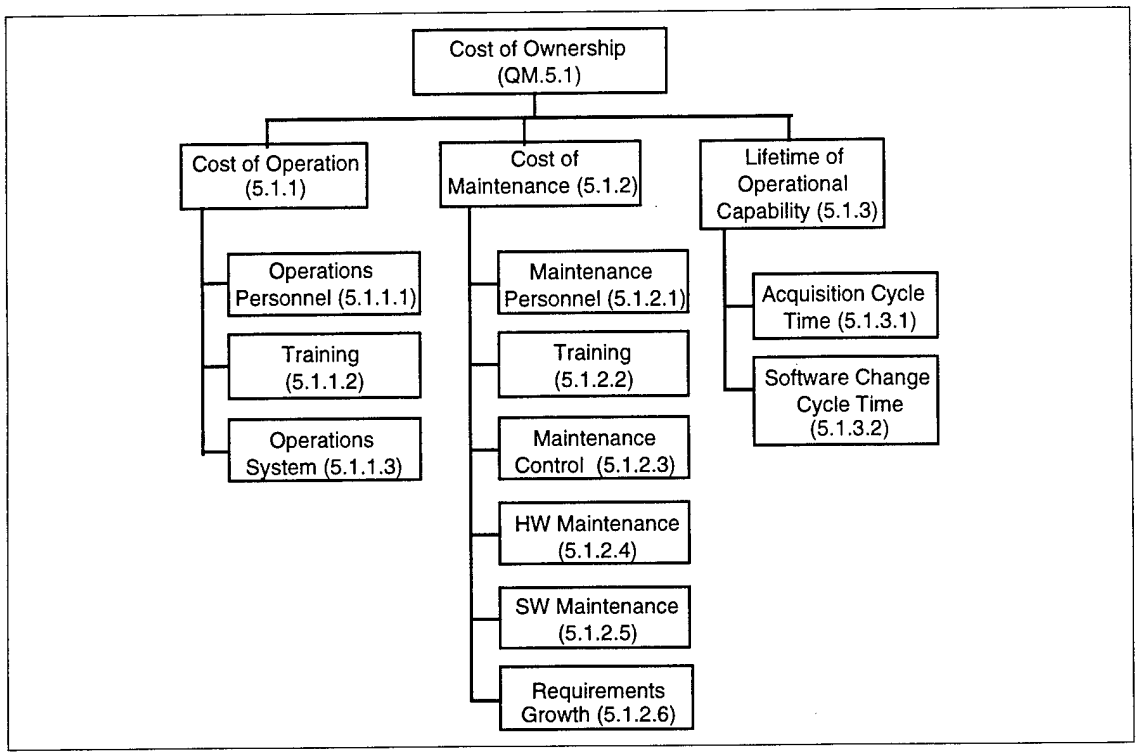


Figure 7: Cost of Ownership

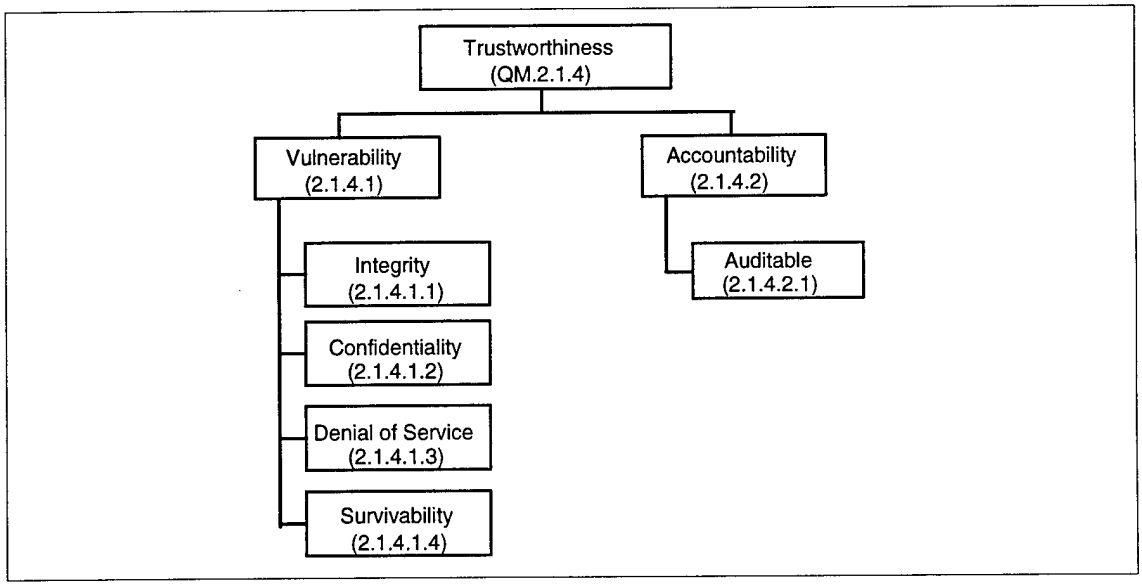


Figure 8: Trustworthiness

2.3.3 Textual Representation

QM. Quality Measures
1. Need Satisfaction Measures
    1. Effectiveness
        1. Necessity of Characteristics
        2. Sufficiency of Characteristics
    2. Responsiveness
    3. Correctness
        1. Completeness/Incompleteness
        2. Consistency
        3. Traceability
        4. Provably Correct
    4. Verifiability
        1. Testability
2. Performance Measures
    1. Dependability
        1. Availability/Robustness
            - Error Tolerance
            - Fault Tolerance
            - Fail Safe
            - Fail Soft
        2. Reliability
            1. Accuracy
        3. Safety
        4. Trustworthiness
            1. Vulnerability
                1. Integrity
                2. Confidentiality
                    - Anonymity
                3. Denial of Service
                    - Accessibility
                4. Survivability
            2. Accountability
                1. Auditable
        5. Security (see also QM.2.1.4)
    2. Efficiency/Resource Utilization
        - Speed
        - Compactness
        1. Capacity
        2. Real-time Responsiveness/Latency
        3. Throughput
    3. Usability
        1. Error Proneness
        2. Operability
    4. Fidelity
3. Maintenance Measures

1. Maintainability
    - Modifiability
    - Flexibility/Adaptability
    - Evolvability/Upgradeability
    - Extendability/Expandability
2. Understandability
    1. Complexity
        - Apparent
        - Inherent
    2. Simplicity
    3. Structuredness
    4. Readability
        - Self-Descriptiveness
        - Conciseness

4. Adaptive Measures
    1. Interoperability
        1. Compatibility
        2. Openness
            - Commonality
    2. Portability
    3. Scalability
    4. Reusability
        1. Functional Scope
            - Generality
            - Abstractness
            - Accessibility
        2. Retrievability

5. Organizational Measures
    1. Cost of Ownership
        1. Cost of Operation
            1. Operations Personnel
            2. Training
            3. Operations system
        2. Cost of maintenance
            1. Maintenance Personnel
            2. Training
            3. Maintenance Control
            4. Hardware Maintenance
            5. Software Maintenance
            6. Requirements Growth
        3. Lifetime of Operational Capability
            1. Acquisition Cycle Time
            2. Software Change Cycle Time
    2. Productivity

2.3.4 Taxonomy-Based Directory to Technology Descriptions

QM. Quality Measures

QM.1 Need Satisfaction Measures

QM.1.1 Effectiveness

QM.1.1.1 Necessity of Characteristics

QM.1.1.2 Sufficiency of Characteristics

QM.1.2 Responsiveness

QM.1.3 Correctness

QM.1.3.1 Completeness/Incompleteness

QM.1.3.2 Consistency

QM.1.3.3 Traceability

QM.1.3.4 Provably Correct

QM.1.4 Verifiability

QM.1.4.1 Testability

QM.2 Performance Measures

QM.2.1 Dependability

QM.3.2 Understandability

QM.5.1.1 Cost of Operation

QM.5.1.1.1 Operations Personnel

QM.5.1.1.2 Training

QM.5.1.1.3 Operations system

QM.5.1.2 Cost of maintenance

QM.5.1.2.1 Maintenance Personnel

QM.5.1.2.2 Training

QM.5.1.2.3 Maintenance Control

QM.5.1.2.4 Hardware Maintenance

QM.5.1.2.5 Software Maintenance

QM.5.1.2.6 Requirements Growth

QM.5.1.3 Lifetime of Operational Capability

QM.5.1.3.1 Acquisition Cycle Time

QM.5.1.3.2 Software Change Cycle Time

QM.5.2 Productivity

3 Technology Descriptions

3.1 Defining Software Technology

This document addresses software technology in its broadest interpretation. Technology is the practical application of scientific knowledge in a particular domain or in a particular manner to accomplish a task. For the purposes of this document, software technology is defined as: the theory and practice of various sciences (to include computer, cognitive, statistical sciences, and others) applied to software development, operation, understanding, and maintenance.

More specifically, we view software technology as any concept, process, method, algorithm, or tool, whose primary purpose is the development, operation, and maintenance of software or software-intensive systems. Technology is not just the technical artifacts, but the knowledge embedded in those artifacts and the knowledge required for their effective use. Software technology may include the following:

- Technology directly used in operational systems, for example: two tier/three tier software architectures, public key digital signatures, remote procedure calls (RPCs), rule-based intrusion detection.
- Technology used in tools that produce (or help produce) or maintain operational systems, for example: graphical user interface (GUI) builders, cyclomatic complexity, Ada 95 programming language, technologies for design rationale capture.
- Process technologies that make people more effective in producing and maintaining operational systems and tools by structuring development approaches or enabling analysis of systems/product lines. Examples include: Personal Software Process[1] (PSP) for Module-Level Development, Cleanroom Software Engineering, Domain Engineering and Domain Analysis.

1. Personal Software Process and PSP are service marks of Carnegie Mellon University.

3.2 Technology Categories

To indicate just how broad our definition of software technology is, we identify below the various categories of entries that are found within this document. A technology description will not explicitly identify the category into which its subject falls, but the reader should be able to infer the category from the information in the entry.

- *Elemental Technology.* An elemental technology can (in general) be traced to a single, identifiable theory or concept related to software development, understanding, operation, or maintenance.
- *Composite Technology.* A composite technology is the integration of several elemental technologies. These component technologies each contribute in some substantive way to the overall composite. The component technologies may or may not have separate descriptions— if they do, this is noted in the description of the composite technology.
- *Group of Technologies.* The document treats technologies as a group in three cases, depending on whether or not the technologies within the group are further distinguished and how the technologies differ from one another:
  - The group as a whole has important and distinguishing characteristics that make it worthy of consideration. But the document doesn't distinguish among technologies within the group, because the internal, external, or usage characteristics that distinguish them are unknown, inaccessible, proprietary, insignificant, or irrelevant to the purposes of the document.
  - Sometimes information is necessary to make a decision about whether or not to use any technology within the group, based on common characteristics of the technology group. In such cases, it is prudent to first consider the technologies in the aggregate before looking at individual technologies within the group.
  - Non-competing technologies that nevertheless contribute to the same application area are grouped together into a tutorial that describes how the technologies can be applied in that particular context.

  In any case, we define the group and describe common characteristics of the group. In the case where members within the group are further distinguished (in separate technology descriptions), we provide cross-references to those technologies.
- *Other Software Technology Topics.* There are certain issues of concern that don't fit into the above categories, yet they are important to software technology. These include certain high-level concepts, such as COTS, component based development/integration, and open systems. In descriptions of these topics, we point to (and explain the relationship to) related technologies.

3.3 Template for Technology Descriptions

The purpose of a technology description is to uniquely identify the technology, to characterize the technology in terms of the properties of systems and measures of software quality that it affects, and to point out tradeoffs, benefits, risks and limitations that may arise in various situations of use. Each technology description also provides reference(s) to literature, indications of the current state of the technology, and cross references to related technologies.

Technology descriptions are not meant to be comprehensive— each description provides the reader with enough knowledge to decide whether to investigate the technology further, to find out where to go for more information, and to know what questions to ask in gathering more information.

Typically, technology descriptions range in size from four to six pages, depending on the amount of information available or the maturity of the technology.

Each technology description has a common format; each major section is described below.

Status. Each technology description contains a status indicator in the upper right-hand corner of its first page. This status indicator provides an assessment of the overall quality and maturity of the technology description. One of four indicators is used: Draft, In Review, Advanced, or Complete. For a more detailed description of these states, please see Explanation of Status Indicators on page 49.

Note. This section appears at the beginning of a technology description if prerequisite or follow-on reading is recommended. The prerequisites are usually overviews of the general topic area that establish a context for the different technologies in the area.

Purpose and Origin. This section provides a general description and brief background of the technology. It describes what capability or benefit was anticipated for the technology when originally conceived. It cites quality measures that are significantly influenced by the technology (these quality measures are italicized), and it identifies common aliases for the technology as well as its originator(s) or key developer(s) (if known).

Technical Detail. This section answers— succinctly— the question "what does the technology do?" It describes the salient quality measures (see the Quality Measures Taxonomy Taxonomy in Section 2.3) that are influenced by the technology in all situations and describes the tradeoffs that are enabled by the technology. It may also provide some insight into why the technology works and what advances are expected. Since the document is not a "how-to" manual, no implementation details are provided.

Usage Considerations. This section provides a context for the use of the technology. Issues that are addressed include

- example applications into which this technology may be incorporated (or should not be incorporated); for instance, "this technology, because of its emphasis on synchronized processing, is particularly suited for real-time applications"
- quality measures that may be influenced by this technology, depending on the particular context in which the application is employed

Maturity. The purpose of this section is to provide an indication as to how well-developed the technology is. A technology that was developed a year or two ago and is still in the experimental stage (or still being developed at the university research level) will likely be more difficult to adopt than one that has been in use in many systems for a decade. It is *not* the intent of this document to provide an absolute measure of maturity, but to provide enough information *to allow the reader to make an informed judgment as to the technology's maturity for their application area.* Details that will help in this determination include

- the extent to which the technology has been incorporated into real systems, tools, or commercial products
- the success that developers have had in adopting and using the technology
- notable failures of the technology (if any)

Other information that might appear in this section includes trend information, such as a projection of the technology's long term potential, observations about the rate of maturation, and implications of rapid maturation.

Costs and Limitations. No technology is right for every situation, and each technology has associated costs (monetary and otherwise). This section points out these limitations and costs. Adopting a technology may limit the use of the application (for instance, it might impose an otherwise unnecessary interface standard). It might require investment in other technologies (see "Dependencies" below as well). It might require investment of time or money. This particular technology may directly conflict with security or real-time requirements. These are just some examples of the kind of limitations that a technology may possess or the costs that it might impose. Specific items of discussion include

- what is needed to adopt this technology (this could mean training requirements, skill levels needed, programming languages, or specific architectures)
- how long it takes to incorporate or implement this technology
- barriers to the use of this technology
- reasons why this technology would not be used

Dependencies. This section identifies other technologies that influence or are influenced by the technology being described. The only dependencies mentioned are those where signifi-

cant influence in either direction is expected. An indication as to why the dependency exists (usually in terms of quality measure or usage consideration) is also provided. If the dependent technology appears in the document, a cross-reference is provided. This paragraph is omitted if no dependencies are known.

Alternatives. An alternative technology is one that could be used for the same purposes as the technology being described. A technology is an alternative if there is any situation or purpose for which both technologies are viable or likely to be considered candidates. Alternatives may represent a simple choice among technologies that achieve the same solution to a problem, or they may represent completely different approaches to the problem being addressed by the technology.

For each alternative technology, this section provides a concise description of the situations for which it provides an alternative. Also provided are any special considerations that could help in selecting among alternatives. If the alternative technology appears in the document, a cross-reference is provided.

Alternative technologies are distinct from dependent or complementary technologies, which must be used in combination with the technology being described to achieve the given purpose.

Complementary Technologies. A complementary technology is one that enhances or is enhanced by the technology being described, but for which neither is critical to the development or use of the other (if it were critical, then it would appear in the "Dependencies" section above). Typically, a complementary technology is one that in combination with this technology will achieve benefits or capabilities that are not obvious when the technologies are considered separately. For each complementary technology, this section provides a concise description of the conditions under which it is complementary and the additional benefits that are provided by the combination. If the complementary technology appears in the document, a cross-reference is provided.

Index Categories. This section provides keywords on which this technology may be indexed. Beside providing the name of the technology, it provides keywords in the following categories:

- *Application category.* This category refers to how this technology would be employed, either in support of operational systems (perhaps in a particular phase of the life cycle) or in actual operation of systems (for example, to provide system security).
- *Quality measures category.* This is a list of those quality attributes (e.g., reliability or responsiveness) that are influenced in some way by the application of this technology.

- Computing Reviews category: This category describes the technical subdiscipline within Computer Science into which the technology falls. The category is based on the *ACM Computing Reviews* Classification System developed in 1991 (and currently undergoing revision). A complete description of the Classification System and its contents can be found in any January issue of *Computing Surveys* or in the annual *ACM Guide to Computing Literature*.

References and Information Sources. The final section in each technology description provides bibliographic information. We include sources cited in the technology description, as well as pointers to key resources that a reader can go to for additional information. These key resources are designated by a check mark (✓). Some care has been taken to choose key references that will best assist one in learning more about the technology.

Author. The author(s) of the technology description are listed in this section. The only exceptions are Draft technology descriptions, which will not have an author's name.

External Reviewer(s). This section contains names of external experts who have reviewed this technology description. If no "External Reviewer(s)" heading is present, then an external review has not occurred.

Last Modified. This is the date on which the technology description was last modified.

Explanation of Status Indicators

Each of the four status indicators is explained below:

Draft technology descriptions have the following attributes:

- They need more work.
- They have generally not been reviewed.
- Overall assessment: While technology descriptions labeled "Draft" will contain some useful information, readers should not rely on these descriptions as their only source of information about the topic. Readers should consider these descriptions as starting points for conducting their own research about the technology.

In Review technology descriptions have the following attributes:

- They are thought to be in fair technical shape.
- They have begun an internal review cycle.
- They may have major issues that must be resolved, or some sections may require additional text.
- Relevant keywords have been added to the Keyword Index.
- Overall assessment: Readers can get some quality information from these, but because these descriptions have not been completely reviewed, readers should explore some of the references for additional information.

Advanced technology descriptions have the following attributes:

- They are in good technical shape.
- Internal review has occurred.
- There are minor issues to be worked, but it is generally polished.
- They are subject to additional review by SEI and external reviewers.
- Relevant keywords have been added to the Keyword Index.
- Overall assessment: These descriptions are in rather good shape, but because they have not been through external review, readers should exercise some caution.
- Note: We encourage readers to critique Advanced technology descriptions, especially for content accuracy. Please see Appendix A, Submitting Information for Subsequent Editions, pg. 407, for more details.

Complete technology descriptions have the following attributes:

- At least one expert external review has occurred, and issues from that review have been resolved.
- Relevant keywords have been added to the Keyword Index.
- No additional work is necessary at this time.
- Overall assessment: These technology descriptions are believed to be complete and correct. They would be revised in the future based on additional external reviewers, new information, and public feedback.
- Note: We encourage readers to critique Complete technology descriptions, especially for content accuracy. Please see Appendix A, Submitting Information for Subsequent Editions, pg. 407, and Appendix B, User Feedback, pg. 409, for more details.

3.4 The Technology Adoption Challenge

Before we move to the actual technology descriptions, it is appropriate to focus on the significant challenges inherent in adopting technology. While individual technology descriptions may contain hints about adopting or inserting a particular technology, this overview of technology adoption focuses on general strategies and guidance applicable to most all situations.

The adoption of a powerful concept, process, method, and/or tool often holds promise of dramatic benefit to an organization. However, efforts to realize these benefits often result in frustration and resistance from those who should receive the benefits. Previous problems with adoption have convinced many to take a very conservative "wait and see" attitude about new technology. Such conservative strategies may reduce the downside, but in today's hyper-competitive world, they may also make it impossible to survive. Mastering the adoption of new technology may indeed separate the winners from the losers.

Experience has shown there are classes of recurring challenges for which effective and repeatable solutions exist. The "trick" is to recognize these recurring challenges and be properly positioned to use one or more of the effective and repeatable solutions. The essence of the "trick" is to realize that the lessons learned by those who have adopted technologies before us may still be relevant. There is much more to be gained in finding how to leverage lessons from the past than from arguing how unique a current situation may be. The key is to realize that any task can usually be decomposed into three types of sub-tasks:

1. recurring tasks for which effective and repeatable solutions are known
2. recurring tasks for which no effective or repeatable solutions are known
3. truly novel tasks

The truly successful organization is one that finds ways to partition a problem to maximize subproblems of the first class and to minimize subproblems of the third. (While problems of the second class are not as desirable as the first, they are better than those of the third; for problems of the second class, we can learn what has been tried before to ensure we don't follow the footsteps of those who have failed.) By leveraging the experience of others, the new technology adopter has freed the organization to focus on the truly novel issues at hand. The rest of this section addresses a collection of issues that are critical for adoption success:

- *Adoption plan.* Plan the adoption as a mission-critical project.
- *Motivation.* Establish the mission-critical motivation for the adoption.
- *Usage design.* Design how the technology will be used to address the mission-critical need.
- *Skill development.* Establish how typical workers and managers will develop the needed skills.
- *Pilot testing.* Pilot test the usage design and the adoption materials.

- *Organizational roll-out.* Roll out the technology to the entire organization after refinements from the pilot effort.
- *Lessons learned.* Continuously learn from usage experience and refine accordingly.

While it is not possible to supply specific solutions to each of these issues, the general nature of the solutions will be provided on the following pages.

The Adoption Plan

Key factors in successful adoption of technology are the quality of the adoption plan and the discipline the organization exhibits to honor the spirit of the plan. The wide variation in the nature of the technologies being adopted and the organizations doing the adoption makes it impossible to recommend a single template for adoption.

An adoption plan should not be viewed as a legal contract between two groups involved in the adoption. Rather, it should be viewed as a communications tool to assist all involved parties as they strive to understand what is to be done, the rationale, and how it is to be accomplished. As a communications tool, an adoption plan must not be viewed as a static document, for insights and technology seldom remain static for long. One finds the right amount of detail to provide in an adoption plan through experience. With too much detail, one risks stifling alternate creative solutions and reducing willingness to alter the plan due to the size of the investment to create it. If there is too little detail, those charged with the implementation may not fully appreciate the intent of the plan or see how the critical aspects of the plan fit together.

An effective adoption considers all of the critical issues listed above, assesses the risks of ignoring these items, and weighs these risks against the costs and potential benefits of addressing and resolving these items. If the adoption of new technology were treated as a mission-critical project and lessons learned from each effort were captured and fed forward to future adoption efforts, the organization would discover that the adoption does not have to be unmanageable or unpredictable.

The primary benefit of an adoption plan is realized only if the plan is used and the organization is prepared to make near-term sacrifices to honor the plan. If the adoption is not mission-critical to the point that management is able to remain focused on its implementation, there is little hope for rapid and effective adoption.

Motivation

One of the first issues that should be captured in the adoption plan is the motivation driving the adoption effort: What is the truly compelling mission need that drives the adoption? If there is no obvious answer to this question, why proceed? Similarly, if the realistic costs associated with the adoption exceed the potential benefit, why proceed? Few adoptions driven by "its the right thing to do" are successful if a compelling motivation is absent.

If a technology adoption is to result in dramatic improvement in how the organization performs its mission, one must assume that dramatic change is required. This implies that a significant number of people in the organization must change the way they perform their jobs. Knowing who must change, what changes are required, and what will motivate them to change is fundamental. The change must be seen as being connected to the mission of the organization in ways the people who must change appreciate and value. Without such a link, fear, uncertainty, and doubt will lead many to resist the change. Most professionals are not motivated well by fear. They respond much more favorably to seeing how their involvement is important and how the success of the effort is directly connected to the success of the organization. Being part of new and critical work that is clearly valued by the most senior people in the organization is often all the motivation required.

Some will try to adopt a new technology as a means of shifting the culture of the organization. For example, if the culture of the organization is to "just do it" without the aid of plans or management oversight, the adoption of a planning tool is unlikely to bring improvement to the organization. If planning discipline is required for organizational survival, you should address the cultural change first and then support the culture change with technology. It seldom works the other way around.

Usage design

When it is clear why the technology must be adopted, the next obvious question is, "How will it be used?" The simplest usage design is the "plug-compatible" replacement— the new technology replaces an existing technology with little dramatic change to the rest of the organization. At the other extreme is complete and radical change, such as a complete reorganization of the organization to support the adoption of the technology.

Experience has shown that plug-compatible replacements can lead to incremental improvements in the organization, but they seldom lead to dramatic improvements: How can an organization perform its mission dramatically better or faster if the majority of the organization doesn't change? This does not mean a plug-compatible approach is bad; rather, it is important to set expectations and ensure the cost/benefit ratio warrants the investment.

Experience has also shown that complete organizational reengineering is a costly path and the real benefits may not be as easy to realize as the advocates might suggest. If a technology demands a complete reorganization before it is possible to benefit from its adoption, other options should be considered unless there are other truly compelling reasons pushing toward reorganization.

Those charged with the success of the adoption need to consider how the organization will function with the new technology in place:

- What roles need to be played?
- What information and work flow is required?
- What interfaces need to exist?
- What management involvement is required?
- How will these map onto the people of the organization, the culture of the organization, and the support process of the organization?

Too many technology adoptions leave these details to each and every affected person to "figure it out" on their own. While most will, there is no guarantee their solutions will be compatible with one another and no guarantee the resulting work flow will be an improvement.

Skill development

Motivated people without the skills to perform their new tasks can lead to real problems. Armed with insight about the new roles, responsibilities, and work flow, the next issue involves assisting the people to develop the skills they will require to be successful. Most people understand the need to "train" the workers in the use of a new tool and this usually translates into some form of class. What often is not appreciated is how much real world practice is required before one becomes skilled in the use of a new tool. It is also common for people to ignore the fact that many skills are perishable. Without regular refreshment, skills fade and disappear.

As indicated earlier, it is critical to recognize all of the various groups in the organization that must change and to question how all of these groups will become skilled in performing their roles. For example

- How will project managers learn to shift how they plan projects and estimate skills and resources during the life of a project due to the use of this technology?
- How will managers track the progress of projects and determine whether adequate progress is being made?
- In case there is a problem, how does one change things without damaging work already performed?
- How do executives assess the return on the investment in a new tool and justify the continued training, sustainment, and overhead costs?

If these questions are not answered, or if the people playing these roles do not understand the answers, the adoption could fail. (Many technically successful adoptions are cancelled because management couldn't appreciate the benefits in terms they valued. This management skill is just as critical as the operator knowing how to use the tool properly.)

Generic classes tend to provide an education, but do little to help students develop the specific skills they will need as they try to use the new tool on the job. There are two rules of thumb:

- The participants will more likely be able to translate their experiences to the job when the training is customized to the organization and skill development activities require hands-on practice.
- The longer the time between the activity and the time when the student is called on to perform, the less likely the first usage will be successful.

If successful first usages are truly critical, one should consider over-staffing these first attempts to minimize the workload placed on each member of the team. It is also helpful to obtain the services of people skilled in the use of the tool and have them play the role of mentor or coach. (Nothing is more comforting than knowing that there is someone there to help when reality does not seem to be the same as what we studied in class or experienced during practice.)

Pilot testing

Pilot testing can be used to reduce organizational risk in the adoption of a new technology. The pilot tests are used to perform two critical tests at the same time:

- determine if the organization can obtain the promised benefits from the technology
- evaluate the adoption approach and materials on a limited scope before taking the technology to the entire organization

Many consider the first test (Can the technology produce the desired benefit?) to be the primary role of pilot testing. In reality, the costs of a pilot test and the impact on the organization make other methods for evaluating the benefit of the technology far more attractive. (For example, sometimes a visit to other real customers would show the benefits at a far lower cost to the organization.) The people running the pilot test should have already determined that the technology can provide value; the only real questions are how hard will it be for this organization to adopt it and benefit from the adoption. A significant caution is that very often pilot projects are not well enough instrumented or baselined to *prove* anything about the value of the technology to the organization; this is a major focus area in any pilot effort.

The best use for pilot tests is to evaluate the adoption of the technology and to showcase the organization's commitment to the changes the technology implies. When senior management embrace the new technology, change their behavior in a visible way to support the technology, and are regularly seen to assist others who are involved with the adoption, the roll-out of the technology will be easier for the rest of the organization. If these leaders are seen as having taken a wait and see attitude, not being intimately involved, and not willing to take risks to make it work, the roll-out is likely to be long and painful.

Selecting the right project for pilot testing is important. The project must be early enough in its life cycle for the team members to be able to develop the skills they will need before being called on to use them. Some things to avoid include

- Retro-fitting parts of a project in order to use it as a pilot test: Most people will view these efforts as a waste of time and effort.
- Picking very short projects, as they are usually not long enough to truly demonstrate the use of the technology.
- Picking a project that is too long, as the results may not be made visible in a timely manner.
- Using a project that is in trouble. The extra effort associated with the pilot test will stress even relatively low-risk projects, and new technologies seldom make up for ill-considered projects, regardless of how effective the technology is at doing what it does.

Some of the most significant benefits of a pilot test are found in determining how to enhance and refine the adoption method based on experience with real people from the organization. Hopefully, the pilot should help in answering the following questions:

- How useful were the skill development activities?
- Could the skill development exercises have been made more realistic?
- Were there vocabulary problems that weren't properly recognized?
- Should the materials be rewritten using nomenclature from the organization?

While these lessons are usually too late to help those doing the pilot test, they can influence what is delivered to the bulk of the organization during roll-out and can significantly reduce the total cost of adoption.

Organizational roll-out

How the organization moves to introduce the technology to the bulk of the organization can dramatically influence the likelihood of success. Force-fitting technology solutions where they don't fit can be disastrous. There are many questions to consider, for example:

- Should the entire organization adopt the technology, or is it really only relevant to part of the organization?
- How long will the technology be used?
- Which groups are at the appropriate places in their project life-cycles for adoption and how long will it be for those who are not yet at reasonable points in their cycle?

A common strategy for organizational roll-out is the mandate. While few mandates have been successful, their failures tend to be due to an unwillingness of the senior leadership to do those things required to make the mandate successful. By means of clear and consistent leadership, personal adoption by the most senior leaders, consistent reward for those who adopt and sanction for those who don't, mandated change is possible. The reality is few executives have the time, energy, or willingness to make the mandate work.

Making adoption optional can only work if there is a clear motivation for projects to risk adoption. If the organizational leadership is willing to fund adoption activities, willing to provide additional technical and managerial resources, and willing to ease external pressures in order to support those who elect to adopt, optional adoptions can be successful. Senior leadership must establish clear motives to support the adoption, honor those who succeed with the adoption, and withhold sanctions from those who honestly tried but failed.

Lessons Learned

The rapidly-changing world will force successful organizations to establish technology adoption as an area of core competence. Just as soon as one technology adoption is completed, management should be considering what the next adoption should be and when it should take place. If the lessons from previous adoptions are not fed forward to the benefit of the subsequent adoptions, the organization is bound to suffer needlessly.

The most powerful tool in technology adoption is the use of experienced people and the use of lessons from previous adoption efforts by the organization. If each adoption effort is performed in a cocoon of ignorance, the team is doomed to repeat previous failures. The organization should record and maintain information that helps answer the following kinds of questions:

- What did we do last time?
- What worked and why?
- What didn't work and why?
- What are we going to do this time and why do we believe these changes will resolve the issues we failed to resolve well before?
- How should we capture and leverage this knowledge so technology adoption does become an area of core competence?

Newton explained his great abilities by asserting that he was "able to stand on the shoulders of giants," those mathematicians who had lain the critical groundwork on which his work was based. Technology adoption can be mastered if we are willing to stand on the shoulders of giants as opposed to insisting that this situation is unique and the past has nothing to tell us.

Conclusion

New technology adoption is like any other major effort an organization tackles. If the organization is dedicated to the task, properly motivated, properly led, properly staffed, properly resourced, and properly managed over the duration of the adoption, the result will most likely be favorable. When leadership is unwilling to make some or all of these commitments, the risks grow and chances are low that the organization will attain benefits from the adoption.

3.5 Alphabetical List of Technology Descriptions

Ada 83

COMPLETE

Purpose and Origin

Ada is a general-purpose, internationally-standardized computer programming language developed by the U.S. Department of Defense to help software designers and programmers develop large, reliable applications. The Ada language enhances *portability*, *maintainability*, *flexibility*, *reliability*, and provides *interoperability* by standardization. The Ada 83 (1983) version [ANSI 83] (international standard: ISO/IEC 8652: 1987) is considered object-based as opposed to object-oriented (see pg. 287) because it does not fully support inheritance or polymorphism [Lawlis 96].

Technical Detail

The Ada language supports principles of good software engineering and discourages poor practices by prohibiting them when possible. Features that support code clarity and encapsulation (use of packages, use of generic packages and subprograms with generic parameters, and private and limited private types) provide support for maintenance and *reusability*. Ada also features strong typing— stronger than C or C++.

The Ada 83 language is independent of any particular hardware or operating system; the interface to any given platform is defined in a specific "System" package. Ada features that support portability include the ability to define numerical types using system-independent declarations and the ability to encapsulate dependencies.

Ada compilers are validated against established written standards— all standard language features exist in every validated Ada compiler. To become validated, a compiler must comply with the Ada Compiler Validation Capability (ACVC) suite of tests. Because of language standardization and required compiler validation, Ada provides an extremely high degree of support for interoperability and portability.

Ada 83 includes features that can be used for object-based programming, but it stops short of providing full support for object-oriented programming (OOP); this is partly because of concerns regarding runtime performance during Ada's development.

By requiring specifications such as type specifications, by performing consistency checks across separately compiled units, and by providing exception handling facilities, Ada 83 provides a high degree of reliability compared to other programming languages.

Ada 83 provides features such as tasking, type declarations, and low-level language features to give explicit support of concurrency and real-time processing. However, Ada 83 does not specify tasking and type declara-

tions in such a way that the resulting performance can always be predicted; this has been a criticism of the language in certain application areas such as embedded, real-time systems.

Usage Considerations

Ada was originally developed to support embedded software systems, but it has proven to provide good support for real-time, computationally-intensive, communication, and information system domains [Lawlis 96].

When combined with static code analysis or formal proofs, Ada can be used in safety-critical systems. For example, Ada has successfully been used in the development of the control systems for the safety-critical Boeing 777 Aircraft [AdaIC 96].

When considering performance, benchmarks performed on both Ada and C software with language toolsets of equal quality and maturity found that the two languages execute equally efficiently— with Ada versions having a slight edge over C versions [Syiek 95]. The quality of the compiled code is determined mostly by the quality of the compiler and not by the language. The burden of optimization is somewhat automated in Ada, as opposed to languages like C, where it is manually performed by the programmer.

When attempting to interface Ada 83 with other languages, several technical issues must be addressed. In order for Ada to call subroutines written in another language, an Ada compiler must support the pragma interface for the other language and its compiler. Similarly, if another language must call Ada subroutines, that language's compiler may also need modifications. The data representation between Ada and the other language must be compatible. Also, the system runtime support environment may need to be modified so that space is not wasted by functionally redundant support software [Hefley 92].

Ada 83 has recently been superseded by Ada 95 (see pg. 67). This new version places the software community into a transition period. Among the issues to be considered in transitioning from Ada 83 to Ada 95 are the following:

- Ada 83 compiler validation status. Validation certificates for all validated Ada 83 compilers expire at the end of March 1998; this may affect maintenance on existing systems written in Ada 83.
- Ada 95 compiler capabilities and availability
- the developmental status of a particular system

The current "philosophy" is that unless a demonstrated need exists, current operational systems or systems currently in development using Ada 83 do not need to transition to Ada 95 [Engle 96]. Refer to the Ada 95

technology description (see pg. 67) for more information on transitioning from Ada 83 to Ada 95.

A significant resource that addresses management and technical issues surrounding the adoption of Ada is the Ada Adoption Handbook [Hefley 92].

Maturity

Ada 83, with over 700 validated compilers [Compilers 96], has been used on a wide variety of programs in embedded, real-time, communication, and information system domains. It is supported by many development environments. Over 4 million lines of Ada code were successfully used in developing the AN/BSY-2 and AN/BQG-5 systems of the Seawolf submarine— a large, extensive, embedded system [Holzer 96]. Ada has become the standard programming language for airborne systems at Boeing Commercial Airplane Group (BCAG). Boeing used Ada to build 60 percent of the systems on the Boeing 777, which represents 70% of the 2.5 million lines of developed code [Pehrson 96, ReuseIC 95].

Ada is increasingly being taught in schools— approximately 323 institutions and companies are teaching Ada— a trend of 25% growth per year in schools and courses; this indicates increased and continued acceptance of Ada as a programming language [AdaIC 96].

Costs and Limitations

In a study performed in 1994, it was found that for life-cycle costs, Ada was twice as cost effective as C [Zeigler 95].

Common perceptions and conventional wisdom regarding Ada 83 (and Ada 95 (see pg. 67)) have been shown to be incorrect or only partially correct. These perceptions include the following:

- Ada is far too complex.
- Ada is too difficult to teach, to learn, to use.
- Ada is too expensive.
- Using Ada causes inefficiencies.
- Training in Ada is too expensive.
- Ada is old-fashioned.
- Ada is not object-oriented.
- Ada does not fit into COTS software.

Mangold examines these perceptions in some detail [Mangold 96].

Alternatives

Other programming languages to consider are Ada 95 (see pg. 67), C, C++, FORTRAN, COBOL, Pascal, Assembly Language, LISP, or Smalltalk.

Index Categories

Name of technology	Ada 83
Application category	Programming Language (AP.1.4.2.1), Compiler (AP.1.4.2.3)
Quality measures category	Reliability (QM.2.1.2), Maintainability (QM.3.1), Interoperability (QM.4.1), Portability (QM.4.2), Scalability (QM.4.3), Reusability (QM.4.4)
Computing reviews category	Programming Languages (D.3)

References and Information Sources

[AdaIC 96] *AdaIC NEWS* [online]. Available WWW <URL: http://sw-eng.falls-church.va.us/home/news/Executive Summary> (1996).

[ANSI 83] ANSI/MIL-STD-1815A-1983. *Reference Manual for the Ada Programming Language.* New York, NY: American National Standards Institute, Inc., 1983.

[Compilers 96] *Ada 83 Validated Compilers List* [online]. Available WWW <URL: http://sw-eng.falls-church.va.us/AdaIC/compilers/83val/83vcl.txt> (August 1996).

[Engle 96] Engle, Chuck. *Re[2]: Ada 83/Ada 95* [email to Gary Haines], [online]. Available email: ghaines@spacecom.af.mil (August 19, 1996).

[Halang 90] Halang, W.A. & Stoyenko, A.D. "Comparative Evaluation of High-Level Real-Time Programming Languages." *Real-Time Systems 2,* 4 (November 1990): 365-82.

[HBAP 96] *Ada Home: The Home of the Brave Ada Programmers (HBAP)* [online]. Available WWW <URL: http://lglwww.epfl.ch:80/Ada/> (1996).

[Hefley 92] Hefley, W.; Foreman, J.; Engle, C.; & Goodenough, J. *Ada Adoption Handbook: A Program Manager's Guide* Version 2.0 (CMU/SEI-92-TR-29, ADA258937). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1992.

[Holzer 96] Holzer, Robert. "Sea Trials Prompt U.S. Navy to Tout Seawolf Sub's Virtues," *Defense News 11,* 28 (July 15-20, 1996): 12.

[Lawlis 96] Lawlis, Patricia K. *Guidelines for Choosing a Computer Language: Support for the Visionary Organization* [online]. Available WWW <URL: http://sw-eng.falls-church.va.us/> (1996).

[Mangold 96] Mangold, K. "Ada95—An Approach to Overcome the Software Crisis?" 4-10. *Proceedings of Ada in Europe 1995.* Frankfurt, Germany, October 2-6, 1995. Berlin, Germany: Springer-Verlag, 1996.

[Pehrson 96] Pehrson, Ron J. *Software Development for the Boeing 777* [online]. Available WWW <URL: http://www.stsc.hill.af.mil/www/xt96jan/xt96d01a.html> (1996).

[Poza 90] Poza, Hugo B. & Cupak Jr., John J. "Ada: The Better Language for Embedded Applications." *Journal of Electronic Defense 13,* 1 (January 1990): 47.

[ReuseIC 95] *Boeing 777: Flying High with Ada and Reuse* [online]. Available WWW <URL: http://sw-eng.falls-church.va.us/ReuseIC/pubs/flyers/boe-reus.htm> (1995).

[Syiek 95] Syiek, David. "C vs. Ada: Arguing Performance Religion." *ACM Ada Letters 15,* 6 (November/December 1995): 67-9.

[Tang 92] Tang, L.S. "A Comparison of Ada and C++," 338-49. *Proceedings of TRI-Ada '92.* Orlando, FL, November 16-20, 1992. New York, NY: Association for Computing Machinery, 1992.

[Zeigler 95] Zeigler, Stephen F. *Comparing Development Costs of C and Ada* [online]. Available WWW <URL: http://sw-eng.falls-church.va.us/> (1995).

Authors Cory Vondrak, TRW, Redondo Beach, CA

Capt Gary Haines, AFMC SSSG
ghaines@spacecom.af.mil

External Reviewer(s) John Goodenough, SEI

Last Modified 10 Jan 97

Ada 95

COMPLETE

Purpose and Origin

Ada is a general-purpose, internationally-standardized computer programming language developed by the U.S. Department of Defense (DoD) to help software designers and programmers develop large, reliable applications. The Ada language enhances *portability*, *maintainability*, *flexibility*, *reliability*, and provides *interoperability* by standardization [Lawlis 96].

The Ada 95 (1995) version [AdaLRM 95] supersedes the 1983 standard (see pg. 61). It corrects some shortcomings uncovered from nearly a decade of using Ada 83, and exploits developments in software technology that were not sufficiently mature at the time of Ada's original design. Specifically, Ada 95 provides extensive support for object-oriented programming (OOP) (see pg. 287), efficient real-time concurrent programming, improved facilities for programming in the large, and increased ability to interface with code written in other languages.

When distinguishing between the two versions of the language, the 1983 version is referred to as Ada 83, and the revised version is referred to as Ada or Ada 95.

Technical Detail

Ada 95 consists of a *core* language that must be supported by all validated compilers, and a set of specialized needs annexes that may or may not be implemented by a specific compiler. However, if a compiler supports a special needs annex, all features of the annex must be supported. The following is the set of annexes [AdaLRM 95]:

Required annexes (i.e., part of core language)

A. Predefined Language Environment
B. Interface to Other Languages
J. Obsolescent Features

Optional special needs annexes

C. Systems Programming
D. Real-time Programming
E. Distributed Systems
F. Information Systems
G. Numerics
H. Safety and Security

Annexes K - P are for informational purposes only and are not part of the standard.

As in Ada 83, Ada 95 compilers are validated against established written standards— all standard language features exist in every validated Ada

compiler. To become validated, a compiler must comply with the Ada Compiler Validation Capability (ACVC) suite of tests [AdaIC 96b]. Because of language standardization and required compiler validation, Ada provides an extremely high degree of support for interoperability and portability.

Like Ada 83, the Ada 95 language is independent of any particular hardware or operating system; the interface to any given platform is defined in a specific "System" package. Ada 95 improves on the Ada 83 features that support portability, which include the ability to define numerical types using system-independent declarations and the ability to encapsulate dependencies.

By requiring specifications such as type specifications, by performing consistency checks across separately compiled units, and by providing exception handling facilities, Ada 95, like Ada 83, provides a high degree of reliability when compared to other programming languages.

The Ada language was developed explicitly to support software engineering— it supports principles of good software engineering and discourages poor practices by prohibiting them where possible. Features supporting code clarity and encapsulation (use of packages, use of generic packages and subprograms with generic parameters, and private and limited private types) provide support for maintenance and *reusability*. Ada 95 also provides full support for object-oriented programming, which allows for a high level of reusability:

- encapsulation of objects and their operations
- OOP inheritance— allowing new abstractions to be built from existing ones by inheriting their properties at either compile time or runtime
- an explicit pointer approach to polymorphism— the programmer must decide to use pointers to represent objects [Brosgol 93]
- dynamic binding

Ada 95 also provides special features (hierarchical libraries and partitions) to assist in the development of very large and distributed software components and systems.

Ada 95 improves on the flexibility provided by Ada 83 for interfacing with other programming languages by better standardizing the interface mechanism and providing an Interface to Other Languages Annex.

Ada 95 improves the specification of previous Ada features that explicitly support concurrency and real-time processing, such as tasking, type declarations, and low-level language features. A Real-Time Program-

ming Annex has been added to better specify the language definition and model for concurrency. Ada 95 has paid careful attention to avoid runtime overhead for the new object-oriented programming (OOP) features and incurs runtime costs commensurate with the generality actually used. Ada 95 also provides the flexibility for the programmer to specify the desired storage reclamation technique that is desired for the application.

Usage Considerations

Ada 95 is essentially an upwardly-compatible extension to Ada 83 with improved support for embedded software systems, real-time systems, computationally-intensive systems, communication systems, and information systems [Lawlis 96]. In revising Ada 83 to Ada 95, incompatibilities were catalogued, tracked, and assessed by the standard revision committee [Taylor 95]. These incompatibilities have proven to be mostly of academic interest, and they have not been a problem in practice.[1]

Combined with at least static code analysis or formal proofs, Ada 95, like Ada 83, is particularly appropriate for use in safety-critical systems.

The Ada Joint Program Office (AJPO) supports Ada 95 by providing an Ada 95 Adoption Handbook [AJPO 95] and an Ada 95 Transition Planning Guide [AJPO 94], and helping form Ada 95 early adoption partnerships with DoD and commercial organizations. The Handbook helps managers understand and assess the transition from Ada 83 to Ada 95 and the Transition Guide is designed to assist managers in developing a transition plan tailored for individual projects [Patton 95]. Another valuable source for Ada 95 training is a multimedia CD-ROM titled *Discovering Ada*. This CD-ROM contains tutorial information, demo programs, and video clips [AdaIC 95].

Ada 95 is the standard programming language for new DoD systems; the use of any other language would require a waiver. Early DoD adoption partnerships who are working Ada 95 projects include the Marine Corps Tactical Systems Support Activity (MCTSSA), Naval Research and Development (NRAD), and the Joint Strike Fighter (JSF) aircraft program [AdaIC 96a].

The AJPO supported the creation of an Ada 95-to-Java J-code compiler. This means that Java (see pg. 221) programs can be created by using Ada. The compiler generates Java "class" files just as a Java language compiler does. Ada and Java components can even call each other [Wheeler 96]. This capability gives Ada, like Java, extensive portability

1. From John Goodenough, SEI, in email to John Foreman, Re: Ada 95, August 16, 1996.

across platforms and allows Internet programmers to take advantage of Ada 95 features unavailable in Java.

Maturity

On February 15, 1995, Ada 95 became the first internationally-standardized object-oriented programming language. As of June 1996, 15 validated compilers were available, with many more expected by the end of the year [Compilers 96]. The Ada 95 compiler validation suite is complete for the core language; Version 2.1, due in March 1997, will provide the capability to validate the additional features in the annexes.

Results from early projects, such as the Joint Automated Message Editing Software (JAMES) and Airfields [AdaIC 96a], indicate that Ada 95 is upwardly-compatible with Ada 83 and that some Ada 95 compilers are mature and stable enough to use on fielded projects [Patton 95]. However, as of the spring of 1996, Ada 95 tool sets and development environments were, in general, still rather immature as compared to Ada 83 versions. Because of the relative immaturity, platform compatibility, bindings (i.e., database, user interface, network interface) availability, and tool support should be closely evaluated when considering Ada 95 compilers.

Costs and Limitations

Common perceptions and conventional wisdom regarding Ada 83 and Ada 95 have been shown to be incorrect or only partially correct. These perceptions include the following:

- Ada is far too complex.
- Ada is too difficult to teach, to learn, to use.
- Ada is too expensive.
- Using Ada causes inefficiencies.
- Training in Ada is too expensive.
- Ada is old-fashioned.
- Ada is not object-oriented.
- Ada does not fit into COTS software.

Mangold examines these perceptions in some detail [Mangold 96].

Alternatives

Other programming languages to consider are Ada 83 (see pg. 61), C, C++, FORTRAN, COBOL, Pascal, Assembly Language, LISP, Smalltalk, or Java (see pg. 221).

Complementary Technologies

The Ada-95-to-Java J-code compiler (discussed in Usage Considerations, pg. 69) enables applications for the Internet to be developed in Ada 95.

Index Categories

Name of technology	Ada 95
Application category	Programming Language (AP.1.4.2.1), Compiler (AP.1.4.2.3)
Quality measures category	Reliability (QM.2.1.2), Maintainability (QM.3.1), Interoperability (QM.4.1), Portability (QM.4.2), Scalability (QM.4.3), Reusability (QM.4.4)
Computing reviews category	Programming Languages (D.3)

References and Information Sources

[AdaLRM 95] *Ada95 Language Reference Manual*, International Standard ISO/IEC 8652: 1995(E), Version 6.0 [online]. Available WWW <URL:http://sw-eng.falls-church.va.us/AdaIC/standards/Welcome.> (1995).

[AdaIC 95] AdaIC News Brief: November 3, 1995 [online]. Available WWW <URL: http://sw-eng.falls-church.va.us/AdaIC/news/weekly/1995/95-11-03.html >(1995).

[AdaIC 96a] *AdaIC NEWS* [online]. Available WWW <URL: http://sw-eng.falls-church.va.us/home/news/Executive Summary> (1996).

[AdaIC 96b] *Validation and Evaluation Test Suites: The Ada compiler certification process* [online]. Available WWW <URL: http://sw-eng.falls-church.va.us/AdaIC/testing/ > (1996).

[AJPO 94] Ada Joint Program Office. *Ada 9X Transition Planning Guide: A Living Document and Working Guide for PEOs and PMs* Version 1.0. Falls Church, VA: Ada Joint Program Office, 1994.

[AJPO 95] Ada Joint Program Office. *Ada 95 Adoption Handbook: A Guide to Investigating Ada 95 Adoption* Version 1.1. Falls Church, VA: Ada Joint Program Office, 1995.

[Brosgol 93] Brosgol, Benjamin. "Object-Oriented Programming in Ada 9X." *Object Magazine 2,* 6 (March-April 1993): 64-65.

[Compilers 96] *Ada 95 Validated Compilers List* [online]. Available WWW <URL: http://sw-eng.falls-church.va.us/AdaIC/compilers/95val/95vcl.txt> (August 1996).

[HBAP 96] *Ada Home: The Home of the Brave Ada Programmers (HBAP)* [online]. Available WWW <URL: http://lglwww.epfl.ch:80/Ada/> (1996).

[Lawlis 96] Lawlis, Patricia K. *Guidelines for Choosing a Computer Language: Support for the Visionary Organization* [online]. Available WWW <URL: http://sw-eng.falls-church.va.us/> (1996).

[Mangold 96] Mangold, K. "Ada95—An Approach to Overcome the Software Crisis?" 4-10. *Proceedings of Ada in Europe 1995.* Frankfurt, Germany, October 2-6, 1995. Berlin, Germany: Springer-Verlag, 1996.

[Patton 95] Patton II, I. Lee. "Early Experiences Adopting Ada 95," 426-34. *Proceedings of TRI-Ada `95.* Anaheim, CA, November 5-10, 1995. New York, NY: Association for Computing Machinery, 1995.

[Taylor 95] Taylor, B. *Ada Compatibility Guide* Version 6.0 [online]. Available WWW <URL: http://sw-eng.falls-church.va.us/AdaIC/docs/compat-guide/compat-guide6-0.txt> (1995).

[Tokar 96] Tokar, Joyce L. "Ada 95: The Language for the 90's and Beyond." *Object Magazine 6,* 4 (June 1996): 53-56.

[Wheeler 96] Wheeler, David A. *Java and Ada* [online]. Available WWW <URL: http://lglsun.epfl.ch/Ada/Tutorials/Lovelace/java.htm> (1996).

Author Cory Vondrak, TRW, Redondo Beach, CA

Capt Gary Haines, AFMC SSSG
ghaines@spacecom.af.mil

External Reviewer(s) Chuck Engle (AJPO director)
John Goodenough, SEI

Last Modified 10 Jan 97

Algorithm Formalization

ADVANCED

Purpose and Origin

In an effort to better understand computer algorithms, researchers in this area began to formally characterize the properties of various classes of algorithms. Initially, research centered on divide-and-conquer and global search algorithms. This initial research proved that these formal algorithm characterizations, called algorithm theories, could be used to synthesize implementations (code) for well-defined functions. Used in program generation or synthesis systems, the purpose of algorithm formalization is two-fold:

- The synthesis of *consistent*, highly CPU efficient algorithms for well-defined functions.
- The formal characterization of algorithm theory notions [Smith 93b]. A by-product of this formalization is the creation of a taxonomy of algorithm theories in which relationships between algorithm theories are formally characterized. These formal characterizations allow a developer to exploit more effectively the structure inherent in the problem space, and thereby allow him to derive or synthesize more *efficient* implementations.

To synthesize an algorithm for a problem using this technology, the essence of the problem and its associated problem domain must be captured in a collection of formal specifications.

Technical Detail

Algorithm synthesis is an emerging correct-by-construction methodology in which algorithm theories are refined to satisfy the constraints represented in an algebraic specification of the problem space [Smith 90]. These algorithm theories represent the structure common to a class of algorithms and abstract out concerns about the specific problem to be solved, the control strategy, the target language and style (e.g., functional versus imperative), and the target architecture. Because theorem provers are used to refine the algorithm theories, the resulting synthesized algorithm is guaranteed to be consistent with the problem specification. In other words, the synthesized algorithm is guaranteed to find solutions to the specified problem provided such solutions exist. If multiple solutions are possible, an algorithm can be synthesized to return one, some, or all of them.

Synthesis systems incorporating formal algorithm theories operate as follows. The developer supplies a formal specification of the problem for which an algorithm is needed, and supplies formal specifications for the operations referenced in the problem specification (i.e., the domain theory). These specifications must be in a prescribed format and language. Using syntactic information drawn from the problem specification, the synthesis system selects candidate algorithm theories from a library of

such theories. The developer selects one of these for refinement. The synthesis system then uses the semantic information provided by the problem and domain theories and— using a theorem prover— completes the refinement process. After the algorithm is generated, a developer will typically apply several computer assisted optimizations to the algorithm before compilation.

Coupling a theorem prover to the algorithm synthesis environment enables computer management of the inherent complexity of the problem and solution spaces, permitting computer management of complex code optimizations. For example, a synthesized algorithm (or implementation) is modified by a user-requested optimization only if the theorem prover is able to verify the consistency of the resulting code. For example, simplification of conditionals, function unfolding (inline expansion), and finite differencing are all possible.

Usage Considerations

The use of this technology encourages reuse of codified knowledge. Specifically, once a domain theory has been developed, it can be used to help define additional problem specifications within that domain, or it can be combined with other domain theories to characterize larger domains. Note, however, that the characterization of large and/or complex domains is non-trivial and may take considerable effort. With respect to the synthesis system itself, a developer is free to add additional algorithm theories to its library. However, the development of such algorithm theories is complex and will require in-depth knowledge of that class of algorithm.

Synthesizing algorithms from formal specifications involves a paradigm shift from traditional programming practice. Because formal specifications are used, developers must formally characterize *what* the operations in the problem domain do rather than stating *how* they do it. In addition, maintenance is not performed on the synthesized code. Instead, the problem specification is modified to reflect the new requirement(s), and an implementation is rederived.

Synthesis of algorithms from formal specifications is independent of the target programming language. However, the synthesis environments themselves may need to be modified to support particular target languages, or code translators may be needed to translate the code generated by the synthesis environment to the desired target language.

Algorithms for non-real time, well-defined deterministic functions— such as sorting or complex scheduling— can be synthesized using this technology. However, additional work is required to determine whether this technology can be extended with notions state and nondeterminism.

Maturity

This technique, along with an algorithm synthesis prototype environment called Kestrel Interactive Development System (KIDS), was developed around 1986 [Smith 86, Smith 91]. Although it initially supported divide-and-conquer and global search algorithm theories, KIDS has been extended with more powerful algorithm theories and with more sophisticated constraint propagation mechanisms. KIDS has been used to synthesize a transportation scheduling algorithm used by US Transportation Command; this scheduling algorithm is able to schedule 10,000 movement requests in approximately one minute, versus hours for competitive scheduling algorithms [Smith 93c]. Ongoing research in this area includes a formalization of local search and formalizations of complex scheduling algorithms. Proof-of-concept scheduling algorithms have been synthesized for the nuclear power-plant domain in which

- scheduled activities can have complex interactions
- timing constraints are represented by earliest start/finish times

This technology is also being extended to address the synthesis of parallel algorithms [Smith 93b].

Costs and Limitations

Like all software development efforts, specification inconsistency may result in implementations that do not meet users' needs. However, the formal nature of problem specifications permits semi-automated investigation of problem specification properties. Adaptation of this technology requires knowledge of discrete mathematics at the level of first order logic and experience in developing formal specifications. Knowledge of constraint propagation, category theory, and resolution-based theorem proving is also required. In addition, formalization of various problem domains may be difficult; to effectively use this technology, special training may be required. However, there are currently no commercially-available, regularly-scheduled courses offered on this subject.

Dependencies

Constraint propagation, resolution-based theorem proving, finite differencing technology (used in verifiably correct optimizations), algebraic specification techniques, and specification construction techniques are enablers for this technology.

Alternatives

Other approaches to developing demonstrably correct algorithm implementations are based on formal verification or deductive synthesis. Software generation systems can be used to select and specialize an algorithm implementation from a library of implementations, to assemble an algorithm for a collection of reusable code fragments, or to generate algorithm implementation stubs (i.e., they can generate code for some parts of an algorithm using syntactic rather than semantic information),

but generally such implementations are not be guaranteed to be consistent with the problem specification.

Complementary Technologies

Category-theoretic specification construction methodologies are useful for developing and refining algorithm, domain, and problem theories. In addition, various domain analysis technologies can be used to investigate the structure of the problem domain.

Index Categories

Name of technology	Algorithm Formalization
Application category	Select or Develop Algorithms (AP.1.3.4)
Quality measures category	Consistency (QM.1.3.2), Provably Correct (QM.1.3.4), Throughput (QM.2.2.3)
Computing reviews category	Algorithms (I.1.2), Automatic Programming (D.1.2), Numerical Algorithms and Problems (F.2.1), Nonumerical Algorithms and Problems (F.2.2), Specifying and Verifying and Reasoning about Programs (F.3.1)

References and Information Sources

[Gomes 96] Gomes, Carla P.; Smith, Douglas; & Westfold, Stephen. "Synthesis of Schedulers for Planned Shutdowns of Power Plants," 12-20. *Proceedings of the Eleventh Knowledge-Based Software Engineering Conference.* Syracuse, NY, September 25-28, 1996. Los Alamitos, CA: IEEE Computer Society Press, 1996.

[Smith 86] Smith, Douglas R. "Top-Down Synthesis of Divide-and-Conquer Algorithms," 35-61. *Readings in Artificial Intelligence and Software Engineering.* Palo Alto, CA: Morgan Kaufmann, 1986.

[Smith 90] Smith, Douglas R. & Lowry, Michael R. "Algorithm Theories and Design Tactics." *Science of Computer Programming 14,* 2-3 (1990): 305-321.

[Smith 91] Smith, Douglas R. "KIDS—A Knowledge-Based Software Development System," 483-514. *Automating Software Design.* Menlo Park, CA: AAAI Press, 1991.

[Smith 93a] Smith, Douglas R. *Classification Approach to Design* (KES.U.93.4). Palo Alto, CA: Kestrel Institute, 1993.

[Smith 93b] Smith, Douglas R. "Derivation of Parallel Sorting Algorithms," 55-69. *Parallel Algorithm Derivation and Program Transformation.* New York, NY: Kluwer Academic Publishers, 1993.

[Smith 93c] Smith, Douglas R. "Transformational Approach to Transportation Scheduling," 60-68. *Proceedings of the Eighth Knowledge-Based Software Engineering Conference*. Chicago, IL, September 20-23, 1993. Los Alamitos, CA: IEEE Computer Society Press, 1993.

Author Mark Gerken, Rome Laboratory
gerken@ai.rl.af.mil

Last Modified 10 Jan 97

Application Programming Interface ADVANCED

Purpose and Origin

Application Programming Interface (API) is an older technology that facilitates exchanging messages or data between two or more different software applications. API is the virtual interface between two interworking software functions, such as a word processor and a spreadsheet. This technology has been expanded from simple subroutine calls to include features that provide for *interoperability* and system *modifiability* in support of the requirement for data sharing between multiple applications. An API is the software that is used to support system-level integration of multiple commercial-off-the-shelf (COTS) software products or newly-developed software into existing or new applications. APIs are also a type of middleware (see pg. 251) that provide for data sharing across different platforms; this is an important feature when developing new or upgrading existing distributed systems. This technology is a way to achieve the total cross-platform consistency that is a goal of open systems (see pg. 135) and standards [Krechmer 92].

Technical Detail

An API is a set of rules for writing function or subroutine calls that access functions in a library. Programs that use these rules or functions in their API calls can communicate with any others that use the API, regardless of the others' specifics [Hines 96]. APIs work with a wide spectrum of application dialogues (i.e., interprogram communication schemes) to facilitate information exchange. These include database access, client/server, peer-to-peer, real-time, event-driven, store and forward, and transaction processing. APIs combine error recovery, data translation, security, queuing, and naming with an easy-to-learn interface that comprises simple but powerful actions/commands (verbs). To invoke an API, a program calls a SEND-type function, specifying parameters for destination name, pointers to the data, and return confirmation options. The API takes the data and does all the communications-specific work transparent to the application.

There are four types of APIs that are enablers of data sharing between different software applications on single or distributed platforms:

- remote procedure calls (RPCs) (see pg. 323)
- Standard Query Language (SQL)
- file transfer
- message delivery

Using RPCs, programs communicate via procedures (or tasks) that act on shared data buffers. SQL is a non-procedural data access language

that allows data sharing between applications by access into a common database. File transfer allows for data sharing by sending formatted files between applications. Message delivery provides data sharing by direct interprogram communications via small formatted messages between loosely- or tightly-coupled applications. Current standards that apply to APIs include the ANSI standard SQL API. There are ongoing efforts to define standards for the other types.

Usage Considerations

APIs can be developed for all computing platforms and operating systems or purchased for most platforms and operating systems. All four API types can be used both on homogeneous and multi-platform applications. However, because of the added complexity required to share data across multiple platforms, RPC, SQL or file transfer APIs are better used to facilitate communication between different applications on homogenous platform systems. These APIs communicate data in different formats (e.g., shared data buffers, database structures, and file constructs). Each data format requires different network commands and parameters to communicate the data properly and can cause many different types of errors. Therefore, in addition to the knowledge required to perform the data sharing tasks, these types of APIs must account for hundreds of network parameters and hundreds of possible error conditions that each application must understand if it is to deliver robust interprogram communications. A message delivery API, in contrast, will offer a smaller subset of commands, network parameters, and error conditions because this API deals with only one format (messages). Because of this reduced complexity, message delivery APIs are a better choice when applications require data sharing across multiple platforms.

Maturity

Many examples of data sharing between different applications have been successfully implemented:

- Covia Technologies, in early 1983, supplied the Communication Integrator (CI), which was the enabler technology for the Apollo airline reservation system used by a consortium of United, British Air, Lufthansa, and other international airlines [King 95].
- DECMessageQ is part of the DECnet infrastructure and has been available since the early 1980s.
- Creative Systems Interface's (CSI) Application to Application Interface (AAI) is a full featured API that is suitable for both client-server and peer-to-peer applications.
- Horizon Strategies' Message Express was initially developed for LU6.2 (IBM generic System Network Architecture protocol) host and VAX/VMS communications. In a typical Message Express manufacturing application, remote plants with VAX, DOS/VSE, and

AS/400 machines conduct work-order scheduling and inventory assessments via peer-to-peer messaging.

Costs and Limitations

APIs may "exist" in many forms; the potential user should comprehend the implications of each. APIs may be

- a bundled part of commercial software packages
- separately-licensed COTS software package(s) (license costs)
- uniquely-developed by a project using the internal capabilities/features of the applications that must communicate

In the last case, which should generally be the exception, the development staff will incur analysis and engineering costs to understand the internal features of the software applications, in addition to the cost to develop and maintain the unique API. In all cases, there are training costs associated with learning how to use the APIs as part of the development and maintenance activity. Additional costs are associated with developing and using APIs to communicate across multiple platforms. As already described, network communications add complexity to the development or use of the APIs. The kinds of costs associated with network applications include additional programming costs, training costs, and licenses for each platform.

Complementary Technologies

APIs can be used in conjunction with the Common Object Request Broker Architecture (see pg. 107), Object Linking and Embedding/Component Object Model (see pg. 271), Distributed Computing Environment (see pg. 167), two-tier architectures (see pg. 381), and three tier architectures (see pg. 367).

Index Categories

Name of technology	Application Programming Interface
Application category	Application Program Interfaces (AP.2.7)
Quality measures category	Maintainability (QM.3.1), Interoperability (QM.4.1)
Computing reviews category	Distributed Systems (C.2.4), Software Engineering Tools and Techniques (D.2.2), Database Management Languages (H.2.3)

References and Information Sources

[Bernstein 96] Bernstein, Philip A. "Middleware: A Model for Distributed Services." *Communications of the ACM 39*, 2 (February 1996): 86-97.

[Hines 96] Hines, John R. "Software Engineering." *IEEE Spectrum* (January 1996): 60-64.

[King 95] King, Steven S. "Message Delivery APIs: The Message is the Medium." *Data Communications 21,* 6 (April 1995): 85-90.

[Krechmer 92] Krechmer, K. "Interface APIs for Wide Area Networks." *Business Communications Review 22,* 11 (November 1992): 72-4.

Author Mike Bray, Lockheed-Martin Ground Systems
michael.w.bray@den.mmc.com

External Reviewer(s) Paul Clements, SEI
John Kereschen, Lockheed Martin Command and Control Systems

Last Modified 10 Jan 97

Architecture Description Languages — COMPLETE

Purpose and Origin

When describing a computer software system, software engineers often talk about the *architecture* of the system, where an architecture is generally considered to consist of components and the connectors (interactions) between them.[1] Although architectural descriptions are playing an increasingly important role in the ability of software engineers to describe and understand software systems, these abstract descriptions are often informal and ad hoc.[2] As a result

- Architectural designs are often poorly understood and not amenable to formal analysis or simulation.
- Architectural design decisions are based more on default than on solid engineering principles.
- Architectural constraints assumed in the initial design are not enforced as the system evolves.
- There are few tools to help the architectural designers with their tasks [Garlan 96].

In an effort to address these problems, formal languages for representing and reasoning about software architecture have been developed. These languages, called architecture description languages (ADLs), seek to increase the *understandability and reusability* of architectural designs, and enable greater degrees of analysis.

Technical Detail

In contrast to module interconnection languages (MILs) (see pg. 255), which only describe the structure of an implemented system, ADLs are used to define and model system architecture *prior* to system implementation. Further, ADLs typically address much more than system structure. In addition to identifying the components and connectors of a system, ADLs typically address:

- *Component behavioral specification.* Unlike MILs, ADLs are concerned with component functionality. ADLs typically provide support for specifying both functional and non-functional

---

1. While definitions of *architecture*, *component*, and *connector* vary among researchers, this definition of architecture serves as a baseline for this technology description. A generally accepted definition describing the difference between a "design" and an "architecture" is that while a design explicitly addresses functional requirements, an architecture explicitly addresses functional and non-functional requirements such as reusability, maintainability, portability, interoperability, testability, efficiency, and fault-tolerance [Paulisch 94].

2. Source: Garlan, David, et al. "ACME: An Architecture Interchange Language." Submitted for publication.

characteristics of components. (Non-functional requirements include those associated with safety, security, reliability, and performance.) Depending on the ADL, timing constraints, properties of component inputs and outputs, and data accuracy may all be specified.

- *Component protocol specification.* Some ADLs, such as Wright [Garlan 94a] and Rapide [Luckham 95], support the specification of relatively complex component communication protocols. Other ADLs, such as UniCon [Shaw 95], allow the type of a component to be specified (e.g., filter, process, etc.) which in turn restricts the type of connector that can be used with it.
- *Connector specification.* ADLs contain structures for specifying properties of connectors, where connectors are used to define interactions between components. In Rapide, connector specifications take the form of partially-ordered event sequences, while in Wright, connector specifications are expressed using Hoare's Communicating Sequential Processes (CSP) language [Hoare 85].

As an example, consider the component shown in Figure 1. This component defines two data types, two operations (op), and an input and an output communication port. The component also includes specifications constraining the behavior of its two operations.

```
Component Simple is
      type in_type is ...
      type out_type is ...
      op f : in_type -> out_type
      op valid_input? : in_type -> Boolean
      port input_port : in_type
      port output_port : out_type
      axiom f-specification is
        (behavioral specification for the operation f)
      end axiom
      axiom valid_input?-specification is
        (behavioral specification for the operation
        valid_input?)
      end axiom
      interface is input_port!(x) ->
        ((output_port!f(x) -> Skip)
        < valid_input?(x) >
        (output_port!(Invalid_Data) -> Skip))
end Simple
```

Figure 1: Component

A protocol specification for this component, written in CSP, defines how it interacts with its environment. Specifically, component Simple will accept a data value x of type in_type on its input port, and, if the data value

is valid, will output f(x) on its output port. If the data value is not valid, Simple will output an error message on its output port. Note that component Simple is a specification, not an implementation. Implementations of ADL components and connectors are expressed in traditional programming languages such as Ada (see pgs. 61 and 67) or C. Facilities for associating implementations with ADL entities vary between ADLs.

Usage Considerations

ADLs were developed to address a need that arose from programming in the large; they are well-suited for representing the architecture of a system or family of systems. Because of this emphasis, several changes to current system development practices may occur:

- *Training.* ADLs are formal, compilable languages that support one or more architectural styles; developers will need training to understand and use ADL technology and architectural concepts/styles effectively (e.g., the use of dataflow, layered, or blackboard architectural styles).
- *Change/emphasis in life-cycle phases.* The paradigm currently used for system development and maintenance may be affected. Specifically, architectural design and analysis will precede code development; results of analysis may be used to alter system architecture. As such, a growing role for ADLs is expected in evaluating competing proposed systems during acquisitions. An ADL specification should provide a good basis for programming activities [Shaw 95].
- *Documentation.* Because the structure of a software system can be explicitly represented in an ADL specification, separate documentation describing software structure is not necessary. This implies that if ADLs are used to define system structure, the architectural documentation of a given system will not become out of date.[1] Additionally, ADLs document system properties in a formal and rigorous way. These formal characterizations can be used to analyze system properties statically and dynamically. For example, dynamic simulation of Rapide [Luckham 95] specifications can be analyzed by automated tools to identify such things as communication bottlenecks and constraint violations. Further, these formal characterizations provide information that can be used to guide reuse.
- *Expanding scope of architecture.* ADLs are not limited to describing the software architecture; application to system architecture (to include hardware, software, and people) is also a significant opportunity.

1. However, one can easily imagine a case where an ADL is used to document the architecture, but then the project moves to the implementation phase and the ADL is forgotten. The code or low-level design migrates, but the architecture is lost. This is often referred to as architectural drift [Perry 92].

Maturity

Several ADLs have been defined and implemented that support a variety of architectural styles, including

- Aesop, which supports the specification and analysis of architectural styles (formal characterizations of common architectures such as pipe and filters, and client-server) [Garlan 94b].
- Rapide, which uses event posets to specify component interfaces and component interaction [Luckham 95].
- Wright, which supports the specification and analysis of communication protocols [Garlan 94a].
- MetaH, which was developed for the real-time avionics domain [Vestal 96].
- LILEAnna, which is designed for use with Ada and generalizes Ada's notion of generics [Tracz 93].
- UniCon, which addresses packaging and functional issues associated with components [Shaw 95].

Further information about these and other languages used to describe software architectures can be found in the *Software Architecture Technology Guide* and *Architectural Description Languages* [SATG 96, SEI 96].

Because ADLs are an emerging technology, there is little evidence in the published literature of successful commercial application. However, Rapide and UniCon have been used on various problems,[1] and MetaH appears to be in use in a commercial setting [Vestal 96]. ADLs often have graphical tools that are similar to CASE tools.

Costs and Limitations

The lack of a common semantic model coupled with differing design goals for various ADLs complicates the ability to share tool suites between them. Researchers are addressing this problem; an ADL called ACME is being developed with the goal that it will serve as an architecture interchange language.[2] Some ADLs, such as MetaH, are domain-specific.

---

1. For example, Rapide has been used to specify/ analyze the architecture model of the Sparc Version 9 64-bit instruction set, a standard published by Sparc International. Models of the extensions for the Ultra Sparc have also been done; they are used extensively in benchmarking Rapide simulation algorithms. Further information is available via the World Wide Web at http://anna.stanford.edu/rapide/rapide.html.

2. Source: Garlan, David, et al. "ACME: An Architecture Interchange Language." Submitted for publication.

In addition, support for asynchronous versus synchronous communication protocols varies between ADLs, as does the ability to express complex component interactions.

Dependencies

Simulation technology is required by those ADLs supporting event-based protocol specification.

Alternatives

The alternatives to ADLs include MILs (see pg. 255) (which only represent the defacto structure of a system), object-oriented CASE tools, and various ad-hoc techniques for representing and reasoning about system architecture.

Another alternative is the use of VHSIC Hardware Description Language (VHDL) tools. While VHDL is often thought of exclusively as a hardware description language, its modularization and communication protocol modeling capabilities are very similar to the ones under development for use in ADLs.

Complementary Technologies

Behavioral specification technologies and their associated theorem proving environments are used by several ADLs to provide capabilities to define component behavior. In addition, formal logics and techniques for representing relationships between them are being used to define mappings between architectures within an ADL and to define mappings between ADLs.

Index Categories

Name of technology	Architecture Description Languages
Application category	Architectural Design (AP.1.3.1), Compiler (AP.1.4.2.3), Plan and Perform Integration (AP.1.4.4)
Quality measures category	Correctness (QM.1.3), Structuredness (QM.3.2.3), Reusability (QM.4.4)
Computing reviews category	Software Engineering Tools and Techniques (D.2.2), Organization and Design (D.4.7), Performance (D.4.8), Systems Programs and Utilities (D.4.9)

References and Information Sources

[Garlan 93] Garlan, David & Shaw, Mary. "An Introduction to Software Architecture," 1-39. *Advances in Software Engineering and Knowledge Engineering* Volume 2. New York, NY: World Scientific Press, 1993.

[Garlan 94a] Garlan, D. & Allen, R. "Formalizing Architectural Connection," 71-80. *Proceedings of the 16th International Conference on Software Engineer-*

ing. Sorrento, Italy, May 16-21, 1994. Los Alamitos, CA: IEEE Computer Society Press, 1994.

[Garlan 94b] Garlan, D.; Allen, R.; & Ockerbloom, J. "Exploiting Style in Architectural Design Environments." *SIGSOFT Software Engineering Notes 19,* 5 (December 1994): 175-188.

[Luckham 95] Luckham, David C., et al. "Specification and Analysis of System Architecture Using Rapide." *IEEE Transactions on Software Engineering 21,* 6 (April 1995): 336-355.

[Hoare 85] Hoare, C.A.R. *Communicating Sequential Processes.* Englewood Cliffs, NJ: Prentice Hall International, 1985.

[Paulisch 94] Paulisch, Frances. "Software Architecture and Reuse— An Inherent Conflict?" 214. *Proceedings of the 3rd International Conference on Software Reuse.* Rio de Janeiro, Brazil, November 1-4, 1994. Los Alamitos, CA: IEEE Computer Society Press, 1994.

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

[SATG 96] *Software Architecture Technology Guide* [online]. Available WWW <URL: http://www-ast.tds-gn.lmco.com/arch/guide.html> (1996).

[SEI 96] *Architectural Description Languages* [online]. Available WWW <URL: http://www.sei.cmu.edu/technology/architecture/adl.html> (1996).

[Shaw 95] Shaw, Mary, et al. "Abstractions for Software Architecture and Tools to Support Them." *IEEE Transactions on Software Engineering 21,* 6 (April 1995): 314-335.

[Shaw 96] Shaw, M. & Garlan, D. *Perspective on an Emerging Discipline: Software Architecture.* Englewood Cliffs, NJ: Prentice Hall, 1996.

[STARS 96] *Scenarios for Analyzing Architecture Description Languages* Version 2.0 [online]. Available WWW <URL: http://www.asset.com/WSRD/abstracts/ABSTRACT_1183.html> (1996).

[Tracz 93] Tracz, W. "LILEANNA: a Parameterized Programming Language," 66-78. *Proceedings of the Second International Workshop on Software Reuse.* Lucca, Italy, March 24-26, 1993. Los Alamitos, CA: IEEE Computer Society Press, 1993.

[Vestal 93] Vestal, Steve. *A Cursory Overview and Comparison of Four Architecture Description Languages* [online]. Available FTP <URL: ftp://ftp.htc.honeywell.com/pub/dssa/papers/four_adl.ps> (1996).

[Vestal 96] Vestal, Steve. *Languages and Tools for Embedded Software Architectures* [online]. Available WWW <URL: http://www.htc.honeywell.com/projects/dssa/dssa_tools.html> (1996).

Author Mark Gerken, Rome Laboratory
gerken@ai.rl.af.mil

External Reviewer(s) Paul Clements, SEI
Paul Kogut, Lockheed Martin, Paoli, PA
Will Tracz, Lockheed Martin Federal Systems, Owego, NY

Last Modified 10 Jan 97

Argument-Based Design Rationale Capture Methods for Requirements Tracing

ADVANCED

Purpose and Origin

A design rationale is a representation of the reasoning behind the design of an artifact. The purpose of argument-based design rationale capturing methods is to track

- the discussions and deliberations that occur during initial requirements analysis
- the reasons behind design decisions
- the changes in the system over the course of its life, whether they are changes in requirements, design, or code (i.e., any software artifact)
- the reasons for and impact of the changes on the system

Replaying the history of design decisions facilitates the understanding of the evolution of the system, identifies decision points in the design phase where alternative decisions could lead to different solutions, and identifies dead-end solution paths. The captured knowledge should enhance the *evolvability* of the system.

The study of argument-based design rationale capture originated during the late 1950s and early 1960s with D. Englebart, who developed a conceptual framework called Humans Using Language, Artifacts, and Methodology in which they are Trained (H-LAM/T) and with Stephen Toulmin and his work concerning the representational form for arguments [Shum 94].

Technical Detail

There are two general approaches to argument-based design rationale capture, both of which are based upon the entity-relationship paradigm:

1. The Issue Based Information Systems (IBIS) that deals with issues, positions, and arguments for which the emphasis is on recording the argumentation process for a single design [Ramesh 92].
2. The Questions, Options, and Criteria (QOC) notation [Shum 94], for which assessments are relationships between options, and criteria and arguments are used to conduct debate about the status of the entities and relationships.

Decision Representation Language (DRL) combines and extends the two approaches to provide support for computational services like dependency management, precedence management, and plausibility management. All of the approaches provide mechanisms for a breadth-first analytic understanding of issues, thus setting the context for concrete refinement of the design.

All of the information gathered using the above mentioned methods/languages is generally called process knowledge. The process knowledge is cross-referenced to the requirements created during the requirements engineering phase. The entities and relationships provide for the structuring of design problems, and they provide a consistent mechanism for decision making and tracking and communication among team members.

Laboratory and small-scale field experiments have been conducted to determine the utility and effectiveness of design rationale capturing methods. Potential benefits include the following:

- Revision becomes a natural process.
- Design rationale capture methods can help to keep the design meetings on track and help maintain a shared awareness of the meeting's process.
- The design rationale record can help identify interrelated issues that need to be resolved. Related arguments enable team members to prepare for the meeting and lead to a better solution.
- The methods can help originators of ideas understand how they are understood by the rest of the team. Note: More analysis is required before the utility of the methods for communicating understandings is fully demonstrated.

The records can be a valuable resource when it becomes necessary to reanalyze a previous decision. Note: There is no data on how frequently the revisitation is necessary, therefore, the benefits may invalidate the effort necessary to capture the information.

Potential pitfalls include the following:

- Care must be taken to avoid prolonged reflective processes and the extensive analysis of high-level or peripheral issues.
- There may be inconsistencies in categorizing the design rationale information in the database because one person's assumptions may be another person's rationale and yet another person's decision.
- Because of the nature of the semiformal language, the reader may need to be familiar with the design to understand the design rationale as represented.

Usage Considerations

The use of this technology requires the development of a shared, consistent, and coherent requirements traceability policy by a project team. Each of the team members must provide commitment to the policy and procedures. A procedure for overall coordination must be developed. To date, these procedures are project-dependent and there is no consistent

policy. It will require effort to generate and maintain the entities and relationships in the design rationale database for a given system.

Maturity

To date, there is at least one commercially-available tool to support the IBIS notation. The vendor also provides training and support for their tool. Proprietary tools to support the IBIS method are being used on government projects (e.g., a database exists with over 100,000 requirements under management) [Ramesh 92]. Tools to support the other methods are in various prototype stages.

Costs and Limitations

Argument-based design rationale capture methods and supporting tools require additional time and effort throughout the software life cycle. Individuals must generate and maintain the entity relationship diagrams for any and all of the methods. Training is essential to make effective use of the methods.

Dependencies

This technology makes use of entity-relationship modeling as the basis for the methods.

Alternatives

There are several alternative approaches to requirements traceability methods. Examples include: Process Knowledge Method, an extension of the argument-based approach that includes a formal representation to provide two way traceability between requirements and artifacts and facilities for temporal reasoning (i.e., mechanisms to use the captured knowledge), and Feature-Based Design Rationale Capture Method for Requirements Tracing (pg. 181), an approach that is centered around the distinctive features of a system.

Index Categories

Name of technology	Argument-Based Design Rationale Capture Methods for Requirements Tracing
Application category	Requirements Tracing (AP.1.2.3)
Quality measures category	Completeness (QM.1.3.1), Consistency (QM.1.3.2), Traceability (QM.1.3.3), Effectiveness (QM.1.1), Reusability (QM.4.4), Understandability (QM.3.2), Maintainability (QM.3.1)
Computing reviews category	Software Engineering Tools and Techniques (D.2.2), Software Engineering Design (D.2.10), Project and People Management (K.6.1)

References and Information Sources

[Gotel 1995] Gotel, Orlena. *Contribution Structures for Requirements Traceability.* London, England: Department of Computing, Imperial College, 1995.

[Ramesh 92] Ramesh, Balasubramaniam & Dhar, Vasant. "Supporting Systems Development by Capturing Deliberations During Requirements Engineering." *IEEE Transactions on Software Engineering 18,* 6 (June 1992): 498-510.

[Ramesh 95] Ramesh, Bala; Stubbs, Lt Curtis; & Edwards, Michael. "Lessons Learned from Implementing Requirements Traceability." *Crosstalk, Journal of Defense Software Engineering 8,* 4 (April 1995).

[Shum 94] Shum, Buckingham Simon & Hammond, Nick. "Argumentation-Based Design Rationale: What Use at What Cost?" *International Journal of Human-Computer Studies 40,* 4 (April 1994): 603-52.

Author Liz Kean, Rome Laboratory
liz@se.rl.af.mil

Last Modified 10 Jan 97

Cleanroom Software Engineering

COMPLETE

Purpose and Origin

Cleanroom software engineering is an engineering and managerial process for the development of high-quality software with certified *reliability*. Cleanroom was originally developed by Dr. Harlan Mills [Linger 94, Mills 87]. The name "Cleanroom" was taken from the electronics industry, where a physical clean room exists to prevent introduction of defects during hardware fabrication. Cleanroom software engineering reflects the same emphasis on defect prevention rather than defect removal, as well as certification of reliability for the intended environment of use.

Technical Detail

The focus of Cleanroom involves moving from traditional, craft-based software development practices to rigorous, engineering-based practices. Cleanroom software engineering yields software that is correct by mathematically sound design, and software that is certified by statistically-valid testing. Reduced cycle time results from an incremental development strategy and the avoidance of rework.

It is well-documented that *significant* differences in cost are associated with errors found at different stages of the software life cycle. By detecting errors as early as possible, Cleanroom reduces the cost of errors during development and the incidence of failures during operation; thus the overall life cycle cost of software developed under Cleanroom can be expected to be far lower than industry average.

The following ideas form the foundation for Cleanroom-based development:

- *Incremental development under statistical quality control (SQC).* Incremental development as practiced in Cleanroom provides a basis for statistical quality control of the development process. Each increment is a complete iteration of the process, and measures of performance in each increment (feedback) are compared with preestablished standards to determine whether or not the process is "in control." If quality standards are not met, testing of the increment ceases and developers return to the design stage.
- *Software development based on mathematical principles.* In Cleanroom development, a key principle is that a computer program is an expression of a mathematical function. The Box Structure Method is used for specification and design, and functional verification is used to confirm that the design is a correct implementation of the specification. Therefore, the specification must define that function before design and functional verification can begin. Verification of program correctness is performed through team

review based on correctness questions. There is no execution of code prior to its submission for independent testing.

- *Software testing based on statistical principles.* In Cleanroom, software testing is viewed as a statistical experiment. A representative subset of all possible uses of the software is generated, and performance of the subset is used as a basis for conclusions about general operational performance. In other words, a "sample" is used to draw conclusions about a "population." Under a testing protocol that is faithful to the principles of applied statistics, a scientifically valid statement can be made about the expected operational performance of the software in terms of reliability and confidence.

Benefits of Cleanroom include significant improvements in *correctness*, *reliability*, and *understandability*. These benefits usually translate into a reduction in field-experienced product failures, reduced cycle time, ease of maintenance, and longer product life.

Usage Considerations

Cleanroom has been documented to be very effective in new development and reengineering (whole system or major subunits) contexts. The following discussion highlights areas where Cleanroom affects or differs from more conventional practice:

- *Team-based development.* A Cleanroom project team is small, typically six to eight persons, and works in a disciplined fashion to ensure the intellectual control of work in progress. Cleanroom teamwork involves peer review of individual work, but does not supplant individual creativity. Once the system architecture has been established and the interfaces between subunits have been defined, individuals typically work alone on a given system component. Individual designs are working drafts that are then reviewed by the team. In a large project, multiple small teams may be formed, with one for the development of each subsystem, thus enabling concurrent engineering after the top-level architecture has been established.
- *Time allocation across life cycle phases.* Because one of the major objectives of Cleanroom is to prevent errors from occurring, the amount of time spent in the design phase of a Cleanroom development is likely to be greater than the amount of time traditionally devoted to design. Cleanroom, however, is not a more time-consuming development methodology, but its greater emphasis on design and verification often yields that concern. Management understanding and acceptance of this essential point— that quality will be achieved by design rather than through testing— must be reflected in the development schedule. Design and verification will require the greatest portion of the schedule. Testing may begin later and be allocated less time than is ordinarily the case. In large Cleanroom projects, where historical data has enabled comparison

of traditional and Cleanroom development schedules, the Cleanroom schedule has equaled or improved upon the usual development time.

- *Existing organizational practices.* Cleanroom does not preclude using other software engineering techniques as long as they are not incompatible with Cleanroom principles. Implementation of the Cleanroom method can take place in a gradual manner. A pilot project can provide an opportunity to "tune" Cleanroom practices to the local culture, and the new practices can be introduced as pilot results to build confidence among software staff.

Maturity

Initial Cleanroom use within IBM occurred in the mid to late 80s, and project use continues to this day. Defense demonstration projects began approximately 1992. Cleanroom has been used on a variety of commercial and defense projects for which reliability was critically important. Some representative examples include the following:

- IBM COBOL/SF product, which has required only a small fraction of its maintenance budget during its operating history [Hausler 94].
- Ericsson OS-32 operating system project, which had a 70% improvement in development productivity, a 100% improvement in testing productivity, and a testing error rate of 1.0 errors per KLOC (represents all errors found in all testing) [Hausler 94].
- USAF Space Command and Control Architectural Infrastructure (SCAI) STARS[1] Demonstration Project at Peterson Air Force Base in Colorado Springs, CO. In this project, Cleanroom was combined with the TRW (spiral) Ada Process Model and some generated and reused code to produce software at a rate of $30-40 per line of code versus industry averages of $130 per line for software of similar complexity and development timeframe (the size of the application is greater than 300 KLOC) [STARSSCAI 95].
- US Army Cleanroom project in the Tank-automotive and Armaments Command at the U.S. Army Picatinny Arsenal. After seven project increments (approximately 90K lines of code), a 4.2:1 productivity increase and a 20:1 return on investment has been documented [Sherer 96a, Sherer 96b]. This project experienced an overall testing error rate (represents all errors found in all testing) of 0.5 errors/KLOC.

In 1995-1996, tools supporting various aspects of the Cleanroom process became commercially available.

Costs and Limitations

Using Cleanroom to accomplish piecemeal, isolated changes to a system not developed using Cleanroom is not considered an effective use of this technology. Training is required and commercially available. Available courses range from overviews to a detailed focus on particular as-

1. STARS: Software Technology for Adaptable Reliable Systems

pects of Cleanroom. For some training classes, it is most productive if software managers and technical staff take the training together. Managers need a thorough understanding of Cleanroom imperatives, and a core group of practitioners needs sufficient orientation in Cleanroom practices to be able to adapt the process to the local environment (this includes establishing a local design language, local verification standards, etc.).

Complementary Technologies

Cleanroom and object-oriented methods. A study/analysis of Cleanroom and three major object-oriented methods: Booch, Objectory, and Shlaer-Mellor (see Object-Oriented Analysis, pg. 275), found that combining object-oriented methods (known for their focus on reusability) with Cleanroom (with its emphasis on rigor, formalisms, and reliability) can define a process capable of producing results that are not only reusable, but also predictable and of high quality.Thus object-oriented methods can be used for front-end domain analysis and Cleanroom can be used for life-cycle application engineering [Ett 96].

Cleanroom and the Capability Maturity Model.[1] The SEI has defined a Cleanroom Reference Model [Linger 96] in terms of a set of Cleanroom Processes for software management, specification, design, and certification, together with a detailed mapping of Cleanroom to the CMM for Software.[2] The mapping shows that Cleanroom and the CMM are fully compatible and mutually reinforcing.

Index Categories

Name of technology	Cleanroom Software Engineering
Application category	Detailed Design (AP.1.3.5), Component Testing (AP.1.4.3.5), System Testing (AP.1.5.3.1), Performance Testing (AP.1.5.3.5), Reengineering (AP.1.9.5)
Quality measures category	Correctness (QM.1.3), Reliability (QM.2.1.2), Understandability (QM.3.2), Availability (QM.2.1.1), Maintainability (QM.3.1)
Computing reviews category	Software Engineering Design (D.2.10)

1. CMM and Capability Maturity Model are service marks of Carnegie Mellon University [Paulk 93].

2. The document is expected to be complete by the end of 1996. Linger, R.C.; Paulk, M.C.; & Trammel, C.J. *Cleanroom Software Engineering Implementation of the CMM for Software* (CMU/SEI-96-TR-023). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1996.

References and Information Sources

[Cleanroom 96] *Cleanroom Tutorial* [online]. Available WWW <URL: http://source.asset.com/stars/loral/cleanroom/tutorial/cleanroom.html> (1996).

[Ett 96] Ett, William. *A Guide to Integration of Object-Oriented Methods and Cleanroom Software Engineering* [online]. Available WWW <URL: http://www.asset.com/stars/loral/cleanroom/oo/guide.html> (1996).

[Hausler 94] Hausler, P. A.; Linger, R. C.; & Trammel, C. J. "Adopting Cleanroom Software Engineering with a Phased Approach." *IBM Systems Journal 33,* 1 (1994): 89-109.

[Linger 94] Linger, R.C. "Cleanroom Process Model." *IEEE Software 11,* 2 (March 1994): 50-58.

[Linger 96] Linger, R.C. & Trammel, C.J. *Cleanroom Software Engineering Reference Model* (CMU/SEI-96-TR-022). Pittsburgh, PA: Carnegie Mellon University, Software Engineering Institute, 1996.

[Mills 87] Mills, H.; Dyer, M.; & Linger, R. "Cleanroom Software Engineering." *IEEE Software 4,* 5 (September 1987): 19-25.

[Paulk 93] Paulk, M.; Curtis B.; Chrissis, M.; & Weber, C. *Capability Maturity Model for Software* Version 1.1 (CMU/SEI-96-TR-24, ADA263403). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1993.

[Sherer 96a] Sherer, S. W. *Cleanroom Software Engineering— the Picatinny Experience* [online]. Available WWW <URL: http://software.pica.army.mil/cleanroom/cseweb.html> (1996).

[Sherer 96b] Sherer, S.W.; Kouchakdjian, A.; & Arnold, P.G. "Experience Using Cleanroom Software Engineering." *IEEE Software 13,* 3 (May 1996): 69-76.

[STARSSCAI 95] Air Force/STARS Demonstration Project Home Page [online]. Available WWW <URL: http://www.asset.com/stars/afdemo/home.html> (1995).

Author

John Foreman, SEI
jtf@sei.cmu.edu

External Reviewer(s)

Wayne Sherer, US Army Picatinny Arsenal
Dave Ceely, Lockheed Martin, Gaithersburg, MD
Dr. Jesse Poore, President, Software Engineering Technologies (SET)
Rick Linger, SEI

Last Modified 10 Jan 97

Client/Server Software Architectures

ADVANCED

Purpose and Origin

The term client/server was first used in the 1980s in reference to personal computers (PCs) on a network. The actual client/server model started gaining acceptance in the late 1980s. The client/server software architecture is a versatile, message-based and modular infrastructure that is intended to improve *usability*, *flexibility*, *interoperability*, and *scalability* as compared to centralized, mainframe, time sharing computing.

A client is defined as a requester of services and a server is defined as the provider of services. A single machine can be both a client and a server depending on the software configuration. For details on client/server software architectures see Schussel and Edelstein [Schussel 96, Edelstein 94].

This technology description provides a summary of some common client/server architectures and, for completeness, also summarizes mainframe and file sharing architectures. Detailed descriptions for many of the individual architectures are provided elsewhere in the document.

Technical Detail

Mainframe architecture (not a client/server architecture). With mainframe software architectures all intelligence is within the central host computer. Users interact with the host through a terminal that captures keystrokes and sends that information to the host. Mainframe software architectures are not tied to a hardware platform. User interaction can be done using PCs and UNIX workstations. A limitation of mainframe software architectures is that they do not easily support graphical user interfaces (see pg. 205) or access to multiple databases from geographically dispersed sites. In the last few years, mainframes have found a new use as a server in distributed client/server architectures (see pg. 227) [Edelstein 94].

File sharing architecture (not a client/server architecture). The original PC networks were based on file sharing architectures, where the server downloads files from the shared location to the desktop environment. The requested user job is then run (including logic and data) in the desktop environment. File sharing architectures work if shared usage is low, update contention is low, and the volume of data to be transferred is low. In the 1990s, PC LAN (local area network) computing changed because the capacity of the file sharing was strained as the number of online user grew (it can only satisfy about 12 users simultaneously) and graphical user interfaces (GUIs) became popular (making mainframe and terminal displays appear out of date). PCs are now being used in client/server architectures [Schussel 96, Edelstein 94].

Client/server architecture. As a result of the limitations of file sharing architectures, the client/server architecture emerged. This approach introduced a database server to replace the file server. Using a relational database management system (DBMS), user queries could be answered directly. The client/server architecture reduced network traffic by providing a query response rather than total file transfer. It improves multi-user updating through a GUI front end to a shared database. In client/server architectures, remote procedure calls (RPCs) (see pg. 323) or standard query language (SQL) statements are typically used to communicate between the client and server [Schussel 96, Edelstein 94].

The remainder of this write-up provides examples of client/server architectures.

Two tier architectures. With two tier client/server architectures (see pg. 381), the user system interface is usually located in the user's desktop environment and the database management services are usually in a server that is a more powerful machine that services many clients. Processing management is split between the user system interface environment and the database management server environment. The database management server provides stored procedures and triggers. There are a number of software vendors that provide tools to simplify development of applications for the two tier client/server architecture [Schussel 96, Edelstein 94].

The two tier client/server architecture is a good solution for distributed computing when work groups are defined as a dozen to 100 people interacting on a LAN simultaneously. It does have a number of limitations. When the number of users exceeds 100, performance begins to deteriorate. This limitation is a result of the server maintaining a connection via "keep-alive" messages with each client, even when no work is being done. A second limitation of the two tier architecture is that implementation of processing management services using vendor proprietary database procedures restricts flexibility and choice of DBMS for applications. Finally, current implementations of the two tier architecture provide limited flexibility in moving (repartitioning) program functionality from one server to another without manually regenerating procedural code. [Schussel 96, Edelstein 94].

Three tier architectures. The three tier architecture (see pg. 367) (also referred to as the multi-tier architecture) emerged to overcome the limitations of the two tier architecture. In the three tier architecture, a middle tier was added between the user system interface client environment and the database management server environment. There are a variety of

ways of implementing this middle tier, such as transaction processing monitors, message servers, or application servers. The middle tier can perform queuing, application execution, and database staging. For example, if the middle tier provides queuing, the client can deliver its request to the middle layer and disengage because the middle tier will access the data and return the answer to the client. In addition the middle layer adds scheduling and prioritization for work in progress. The three tier client/server architecture has been shown to improve performance for groups with a large number of users (in the thousands) and improves flexibility when compared to the two tier approach. Flexibility in partitioning can be a simple as "dragging and dropping" application code modules onto different computers in some three tier architectures. A limitation with three tier architectures is that the development environment is reportedly more difficult to use than the visually-oriented development of two tier applications [Schussel 96, Edelstein 94]. Recently, mainframes have found a new use as servers in three tier architectures (see pg. 227).

Three tier architecture with transaction processing monitor technology. The most basic type of three tier architecture has a middle layer consisting of Transaction Processing (TP) monitor technology (see pg. 373). The TP monitor technology is a type of message queuing, transaction scheduling, and prioritization service where the client connects to the TP monitor (middle tier) instead of the database server. The transaction is accepted by the monitor, which queues it and then takes responsibility for managing it to completion, thus freeing up the client. When the capability is provided by third party middleware vendors it is referred to as "TP Heavy" because it can service thousands of users. When it is embedded in the DBMS (and could be considered a two tier architecture), it is referred to as "TP Lite" because experience has shown performance degradation when over 100 clients are connected. TP monitor technology also provides

- the ability to update multiple different DBMSs in a single transaction
- connectivity to a variety of data sources including flat files, non-relational DBMS, and the mainframe
- the ability to attach priorities to transactions
- robust security

Using a three tier client/server architecture with TP monitor technology results in an environment that is considerably more scalable than a two tier architecture with direct client to server connection. For systems with thousands of users, TP monitor technology (not embedded in the DBMS) has been reported as one of the most effective solutions. A limitation to TP monitor technology is that the implementation code is usually written

in a lower level language (such as COBOL), and not yet widely available in the popular visual toolsets [Schussel 96].

Three tier with message server. Messaging is another way to implement three tier architectures. Messages are prioritized and processed asynchronously. Messages consist of headers that contain priority information, and the address and identification number. The message server connects to the relational DBMS and other data sources. The difference in TP monitor technology and message server is that the message server architecture focuses on intelligent messages, whereas the TP Monitor environment has the intelligence in the monitor, and treats transactions as dumb data packets. Messaging systems are good solutions for wireless infrastructures [Schussel 96].

Three tier with an application server. The three tier application server architecture allocates the main body of an application to run on a shared host rather than in the user system interface client environment. The application server does not drive the GUIs; rather it shares business logic, computations, and a data retrieval engine. Advantages are that with less software on the client there is less security to worry about, applications are more scalable, and support and installation costs are less on a single server than maintaining each on a desktop client [Schussel 96].

Three tier with an ORB architecture. Currently industry is working on developing standards to improve interoperability and determine what the common object request broker (ORB) (see pg. 291) will be. There are two candidates, OLE (see pg. 271) and CORBA (see pg. 107). It is expected that by 1997 both candidates will be available and have some degree of interoperability. With distributed objects being self-contained and executable (all data and procedures present), distributed object computing holds the promise that features such as fault tolerance may be achieved by just copying objects onto multiple servers. The application server design should be used when security, scalability, and cost are major considerations [Schussel 96].

Distributed/collaborative enterprise architecture. The distributed/collaborative enterprise architecture emerged in 1993 (see pg. 163). This software architecture is based on Object Request Broker (ORB) (see pg. 291) technology, but goes further than the Common Object Request Broker Architecture (CORBA) (see pg. 107) by using shared, reusable business models (not just objects) on an enterprise-wide scale. The benefit of this architectural approach is that standardized business object models and distributed object computing are combined to give an organization flexibility to improve effectiveness organizationally, operationally,

and technologically. An enterprise is defined here as a system comprised of multiple business systems or subsystems. Distributed/collaborative enterprise architectures are limited by a lack of commercially-available object orientation analysis and design method tools that focus on applications [Shelton 93, Adler 95].

Usage Considerations

Client/server architectures are being used throughout industry and the military. They provide a versatile infrastructure that supports insertion of new technology more readily than earlier software designs.

Maturity

Client/server software architectures have been in use since the late 1980s. See individual technology descriptions for more detail.

Costs and Limitations

There a number of tradeoffs that must be made to select the appropriate client/server architecture. These include business strategic planning, and potential growth on the number of users, cost, and the homogeneity of the current and future computational environment.

Dependencies

Developing a client/server architecture following an object-oriented methodology would be dependent on the CORBA or OLE standards for design implementation (see pg. 107 and pg. 271).

Alternatives

Alternatives to client/server architectures would be mainframe or file sharing architectures.

Complementary Technologies

Complementary technologies for client/server architectures are computer-aided software engineering (CASE) tools because they facilitate client/server architectural development, and open systems (see pg. 135) because they facilitate the development of architectures that improve scalability and flexibility.

Index Categories

Name of technology	Client/Server Software Architectures
Application category	Software Architecture Models (AP.2.1.1)
Quality measures category	Usability (QM.2.3), Scalability (QM.4.3), Maintainability (QM.3.1), Interoperability (QM.4.1)
Computing reviews category	Distributed Systems (C.2.4), Software Engineering Design (D.2.10)

References and Information Sources

[Adler 95] Adler, R. M. "Distributed Coordination Models for Client/Sever Computing." *Computer 28,* 4 (April 1995): 14-22.

[Dickman 95] Dickman, A. "Two-Tier Versus Three-Tier Apps." *Informationweek* 553 (November 13, 1995): 74-80.

[Edelstein 94] Edelstein, Herb. "Unraveling Client/Server Architecture." *DBMS 7,* 5 (May 1994): 34(7).

[Gallaugher 96] Gallaugher, J. & Ramanathan, S. "Choosing a Client/Server Architecture. A Comparison of Two-Tier and Three-Tier Systems." *Information Systems Management Magazine 13,* 2 (Spring 1996): 7-13.

[Louis 95] *Louis* [online]. Available WWW <URL: http://www.softis.is> (1995).

[Newell 95] Newell, D.; Jones, O.; & Machura, M. "Interoperable Object Models for Large Scale Distributed Systems," 30-31. *Proceedings. International Seminar on Client/Server Computing.* La Hulpe, Belgium, October 30-31, 1995. London, England: IEE, 1995.

[Schussel 96] Schussel, George. *Client/Server Past, Present, and Future* [online]. Available WWW <URL: http://www.dciexpo.com/geos/> (1995).

[Shelton 93] Shelton, Robert E. "The Distributed Enterprise (Shared, Reusable Business Models the Next Step in Distributed Object Computing)." *Distributed Computing Monitor 8,* 10 (October 1993): 1.

Author Darleen Sadoski, GTE
sadoski.darleen@mail.ndhm.gtegsc.com

External Reviewer(s) Frank Rogers, GTE

Last Modified 10 Jan 97

Common Object Request Broker Architecture ADVANCED

Note

We recommend Object Request Broker, pg. 291, as prerequisite reading for this technology description.

Purpose and Origin

The Common Object Request Broker Architecture (CORBA) is a *specification* of a standard architecture for object request brokers (ORBs) (see pg. 291). A standard architecture allows vendors to develop ORB products that support application *portability* and *interoperability* across different programming languages, hardware platforms, operating systems, and ORB implementations:

> "Using a CORBA-compliant ORB, a client can transparently invoke a method on a server object, which can be on the same machine or across a network. The ORB intercepts the call, and is responsible for finding an object that can implement the request, passing it the parameters, invoking its method, and returning the results of the invocation. The client does not have to be aware of where the object is located, its programming language, its operating system or any other aspects that are not part of an object's interface" [OMG 96].

The "vision" behind CORBA is that distributed systems are conceived and implemented as distributed objects. The interfaces to these objects are described in a high-level, architecture-neutral specification language that also supports object-oriented design abstraction. When combined with the Object Management Architecture (see Technical Detail, pg. 108), CORBA can result in distributed systems that can be rapidly developed, and can reap the benefits that result from using high-level building blocks provided by CORBA, such as *maintainability* and *adaptability*.

The CORBA specification was developed by the Object Management Group (OMG), an industry group with over six hundred member companies representing computer manufacturers, independent software vendors, and a variety of government and academic organizations [OMG 96]. Thus, CORBA specifies an industry/consortium standard, not a "formal" standard in the IEEE/ANSI/ISO sense of the term. The OMG was established in 1988, and the initial CORBA specification emerged in 1992. Since then, the CORBA specification has undergone significant revision, with the latest major revision (CORBA v2.0) released in July 1996.

Technical Detail

CORBA ORBs are middleware mechanisms (see pg. 251), as are all ORBs. CORBA can be thought of as a generalization of remote procedure call (RPC) that includes a number of refinements of RPC, including:

- a more abstract and powerful interface definition language
- direct support for a variety of object-oriented concepts
- a variety of other improvements and generalizations of the more primitive RPC

CORBA and the Object Management Architecture. It is impossible to understand CORBA without appreciating its role in the Object Management Architecture (OMA), shown in Figure 2. The OMA is itself a specification (actually, a collection of related specifications) that defines a broad range of services for building distributed applications. The OMA goes far beyond RPC in scope and complexity. The distinction between CORBA and the OMA is an important one because many services one might expect to find in a middleware product such as CORBA (e.g., naming, transaction, and asynchronous event management services) are actually specified as services in the OMA. For reference, the OMA reference architecture encompasses both the ORB and remote service/object depicted in Figure 21, pg. 291.

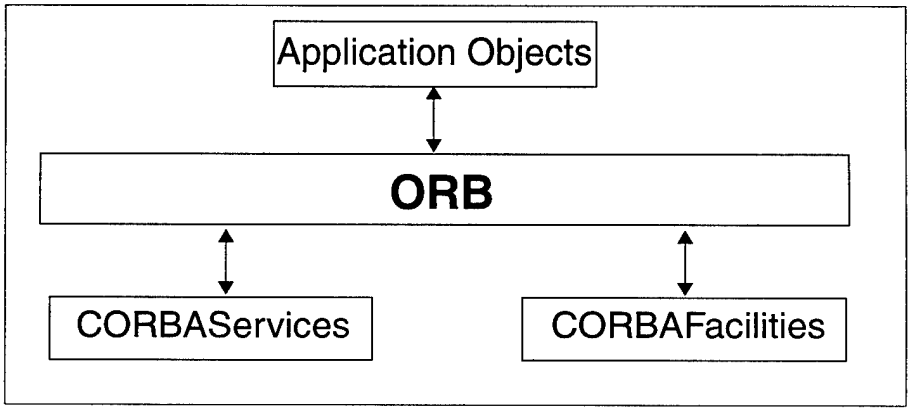


Figure 2: Object Management Architecture

OMA services are partitioned into three categories: CORBAServices, CORBAFacilities, and ApplicationObjects. The ORB (whose details are specified by CORBA) is a communication infrastructure through which applications access these services, and through which objects interact with each other. CORBAServices, CORBAFacilities, and ApplicationObjects define different categories of objects in the OMA; these objects

(more accurately object *types*) define a range of functionality needed to support the development of distributed software systems.

- CORBAServices are considered fundamental to building non-trivial distributed applications. These services currently include asynchronous event management, transactions, persistence, externalization, concurrency, naming, relationships, and lifecycle. Table 1 summarizes the purpose of each of these services.
- CORBAFacilities may be useful for distributed applications in some settings, but are not considered as universally applicable as CORBAServices. These "facilities" include: user interface, information management, system management, task management, and a variety of "vertical market" facilities in domains such as manufacturing, distributed simulation, and accounting.
- Application Objects provide services that are particular to an application or class of applications. These are not (currently) a topic for standardization within the OMA, but are usually included in the OMA reference model for completeness, i.e., objects are either application-specific, support common facilities, or are basic services.

Table 1: Overview of CORBA Services

Naming Service	Provides the ability to bind a name to an object. Similar to other forms of directory service.
Event Service	Supports asynchronous message-based communication among objects. Supports chaining of event channels, and a variety of producer/consumer roles.
Lifecycle Service	Defines conventions for creating, deleting, copying and moving objects.
Persistence Service	Provides a means for retaining and managing the persistent state of objects.
Transaction Service	Supports multiple transaction models, including mandatory "flat" and optional "nested" transactions.
Concurrency Service	Supports concurrent, coordinated access to objects from multiple clients.
Relationship Service	Supports the specification, creation and maintenance of relationships among objects.
Externalization Service	Defines protocols and conventions for externalizing and internalizing objects across processes and across ORBs.

CORBA in detail. Figure 3 depicts most of the basic components and interfaces defined by CORBA. This figure is an expansion of the ORB component of the OMA depicted in Figure 2.

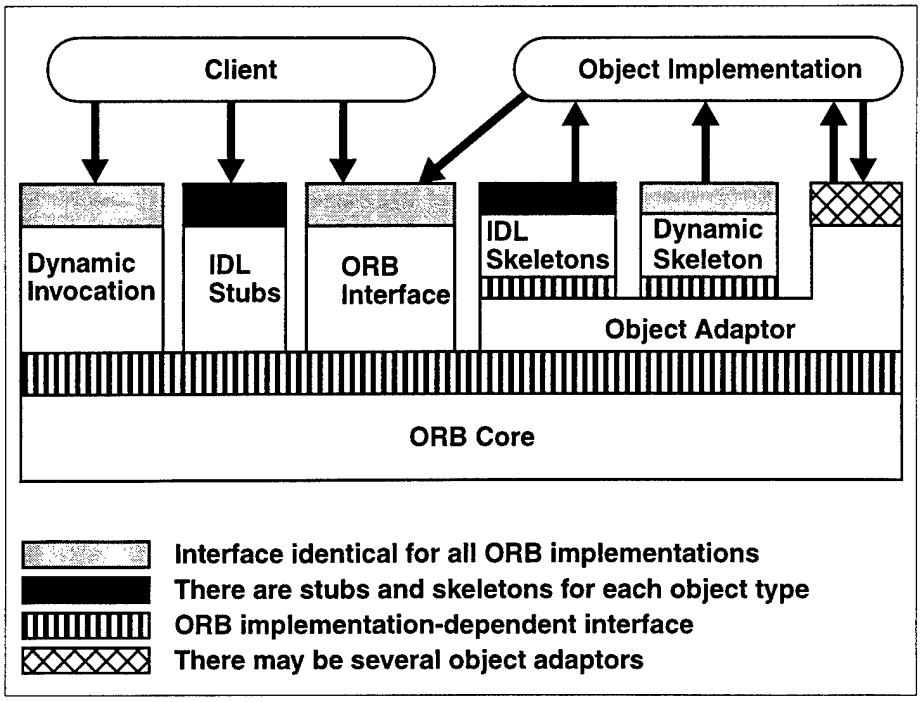


Figure 3: Structure of CORBA Interfaces

One element (not depicted in Figure 2) that is crucial to the understanding of CORBA is the interface definition language (IDL) processor. All objects are defined in CORBA (actually, in the OMA) using IDL. IDL is an object-oriented interface definition formalism that has some syntactic similarities with C++. Unlike C++, IDL can only define interfaces; it is not possible to specify behavior in IDL. Language mappings are defined from IDL to C, C++, Ada95, and Smalltalk80.

An important point to note is that CORBA specifies that clients and object implementations can be written in different programming languages and execute on different computer hardware architectures and different operating systems, and that clients and object implementations can not detect any of these details about each other. Put another way, the IDL interface completely defines the interface between clients and objects; all other details about objects (such as their implementation language and location) can be made "transparent."

Table 2 summarizes the components of CORBA and their functional role.

Table 2: Components of the CORBA Specification

ORB Core	The CORBA runtime infrastructure. The interface to the ORB Core is not defined by CORBA, and will be vendor proprietary.
ORB Interface	A standard interface (defined in IDL) to functions provided by all CORBA-compliant ORBs.
IDL Stubs	Generated by the IDL processor for each interface defined in IDL. Stubs hide the low-level networking details of object communication from the client, while presenting a high-level, object type-specific application programming interface (API).
Dynamic Invocation Interface (DII)	An alternative to stubs for clients to access objects. While stubs provide an object type-specific API, DII provides a generic mechanism for constructing requests at run time (hence "dynamic invocation"). An interface repository (another CORBA component not illustrated in Figure 2) allows some measure of type checking to ensure that a target object can support the request made by the client.
Object Adaptor	Provides extensibility of CORBA-compliant ORBs to integrate alternative object technologies into the OMA. For example, adaptors may be developed to allow remote access to objects that are stored in an object-oriented database. Each CORBA-compliant ORB must support a specific object adaptor called the Basic Object Adaptor (BOA) (not illustrated in Figure 2). The BOA defines a standard API implemented by all ORBs.
IDL Skeletons	The server-side (or object implementation-side) analogue of IDL stubs. IDL skeletons receive requests for services from the object adaptor, and call the appropriate operations in the object implementation.
Dynamic Skeleton Interface (DSI)	The server-side (or object implementation-side) analogue of the DII. While IDL skeletons invoke specific operations in the object implementation, DSI defers this processing to the object implementation. This is useful for developing bridges and other mechanisms to support inter-ORB interoperation.

Usage Considerations

Compliance. As noted, CORBA is a specification, not an implementation. Therefore, the question of compliance is important: How does a consumer know if a product is CORBA-compliant, and, if so, what does that mean? CORBA compliance is defined by the OMG:

> "The minimum required for a CORBA-compliant system is adherence to the specifications in CORBA Core and one mapping" [CORBA 96]

where "mapping" refers to a mapping from IDL to a programming language (C, C++ or Smalltalk80; Ada95 is specified but has not been formally adopted by the OMG at the time of this writing). The CORBA Core (*not* the same as the ORB Core denoted in Figure 3 and Table 2) is defined for compliance as including the following:

- the interfaces to all of the elements depicted in Figure 3
- interfaces to the interface repository (not shown in Figure 3)
- a definition of IDL syntax and semantics
- the definition of the object model that underlies CORBA (e.g., what is an object, how is it defined, where do they come from)

Significantly, the CORBA Core does *not* include CORBA interoperability, nor does it include interworking, the term used to describe how CORBA is intended to work with Microsoft's COM (see pg. 291). A separate but related point is that CORBA ORBs need not provide implementations of any OMA services.

There are as yet no defined test suites for assessing CORBA compliance. Users must evaluate vendor claims on face value, and assess the likelihood of vendor compliance based upon a variety of imponderables, such as the role played by the vendor in the OMG; vendor market share; and press releases and testimonials. Hands-on evaluation of ORB products is an absolute necessity. However, given the lack of a predefined compliance test suite, the complexity of the CORBA specification (see next topic), and the variability of vendor implementation choices, even this will be inadequate to fully assess "compliance."

Although not concerned with compliance testing in a formal sense, one organization has developed an operational testbed for demonstrating ORB interoperability [CORBANet 96]. It is conceivable that other similar centers may be developed that address different aspects of CORBA (e.g., real time, security), or that do formal compliance testing. However, no such centers exist at the time of this writing.

Complexity. CORBA is a complex specification, and considerable effort may be required to develop expertise in its use. A number of factors compound the inherent complexity of the CORBA specification.

- While CORBA defines a standard, there is great latitude in many of the implementation details— ORBs developed by different vendors may have significantly different features and capabilities. Thus, users must learn a specification, the way vendors implement the

specification, and their value-added features (which are often necessary to make a CORBA product usable).

- While CORBA makes the development of distributed applications easier than with previous technologies, this ease of use may be deceptive: The difficult issues involved in designing robust distributed systems still remain (e.g., performance prediction and analysis, failure mode analysis, consistency and caching, and security).
- Facility with CORBA may require deep expertise in related technologies, such as distributed systems design, distributed and multi-threaded programming and debugging; inter-networking; object-oriented design, analysis, and programming. In particular, expertise in object-oriented technology may require a substantial change in engineering practice, with all the technology transition issues that implies (see The Technology Adoption Challenge, pg. 51).

Stability. CORBA (and the OMA) represent a classical model of distributed computing, despite the addition of object-oriented abstraction. Recent advances in distributed computing have altered the landscape CORBA occupies. Specifically, the recent emergence of mobile objects via Java (see pg. 221), and the connection of Java with "web browser" technologies has muddied the waters concerning the role of CORBA in future distributed systems. CORBA vendors are responding by supporting the development of "ORBlets", i.e., Java applets that invoke the services of remote CORBA objects. However, recent additions to Java support remote object invocation directly in a native Java form. The upshot is that, at the time of this writing, there is great instability in the distributed object technology marketplace.

Industry standards such as CORBA have the advantage of flexibility in response to changes in market conditions and technology advances (in comparison, formal standards bodies move much more slowly). On the other hand, changes to the CORBA specifications— while technically justified— have resulted in unstable ORB implementations. For example, CORBA v2.0, released in July 1995 with revisions in July 1996, introduced features to support interoperation among different vendor ORBs. These features are not yet universally available in all CORBA ORBs, and those ORBs that implement these features do so in uneven ways. Although the situation regarding interoperation among CORBA ORBs is improving, instability of implementations is the price paid for flexibility and evolvability of specification.

The OMA is also evolving, and different aspects are at different maturity levels. For instance, CORBAFacilities defines more of a framework for desired services than a specification suitable for implementation. The more fundamental CORBAServices, while better defined, are not rigor-

ously defined; a potential consequence is that different vendor implementations of these services may differ widely both in performance and in semantics. The consequence is particularly troubling in light of the new interoperability features; prior to inter-ORB interoperability the lack of uniformity among CORBAServices implementations would not have been an issue.

Maturity

A large and growing number of implementations of CORBA are available in the marketplace, including implementations from most major computer manufacturers and independent software vendors. See pg. 293 for a listing of available CORBA-compliant ORBs. CORBA ORBs are also being developed by university research and development projects, for example Stanford's Fresco, XeroxPARC's ILU, Cornell's Electra, and others.

At the same time, it must be noted that not all CORBA ORBs are equally mature, nor has the OMA sufficiently matured to support the vision that lies behind CORBA (see Purpose and Origin, pg. 107). While CORBA and OMA products are maturing and are being used in increasingly complex and demanding situations, the specifications and product implementations are not entirely stable. This is in no small way a result of the dynamism of distributed object technology and middleware in general and is no particular fault of the OMG. Fortunately techniques exist for evaluating technology in the face of such dynamism [Wallace 96, Brown 96].

Costs and Limitations

Costs and limitations include the following:

- CORBA v2.0 does not address real-time issues.
- IDL is a "least-common denominator" language. It does not fully exploit the capabilities of programming languages to which it is mapped, especially where the definition of abstract types is concerned.
- The price of ORBs varies greatly, from a few hundred to several thousand dollars. Licensing schemes also vary.
- Training is essential for the already experienced programmer: five days of hands-on training for CORBA programming fundamentals is suggested [Mowbray 93].
- CORBA specifies only a minimal range of security mechanisms; more ambitious and comprehensive mechanisms have not yet been adopted by the OMG. Deng discusses the potential integration of security into CORBA-based systems [Deng 95].

Dependencies

Dependencies include the following:

- TCP/IP is needed to support the CORBA-defined inter-ORB interoperability protocol (IIOP).
- Most commercial CORBA ORBs rely on C++ as the principal client and server programming environment. Java-specific ORBs are also emerging.

Alternatives

Alternatives include the following:

- The Open Group's Distributed Computing Environment (DCE) is sometimes cited as an alternative "open" specification for distributed computing (see pg. 167).
- Where openness is not a concern and "Wintel" platforms are dominant, Microsoft's COM/DCOM may be suitable alternatives.
- Other middleware technologies may be appropriate in different settings (e.g., message-oriented middleware (see pg. 247)).

Complementary Technologies

Complementary technologies include the following:

- Java and/or web browsers can be used in conjunction with CORBA, although precise usage patterns have not yet emerged and are still highly volatile.
- Object-oriented database management systems (OODBMS) vendors are developing object adaptors to support more robust 3-tier architecture (see pg. 367) development using CORBA.

Index Categories

Name of technology	Common Object Request Broker Architecture
Application category	Client/Server (AP.2.1.2.1), Client/Server Communication (AP.2.2.1)
Quality measures category	Maintainability (QM.3.1), Interoperability (QM.4.1), Portability (QM.4.2), Scalability (QM.4.3), Reusability (QM.4.4)
Computing reviews category	Distributed Systems (C.2.4), Object-Oriented Programming (D.1.5)

References and Information Sources

[Baker 94] Baker, S. "CORBA Implementation Issues." *IEEE Colloquium on Distributed Object Management Digest 1994* 7 (January 1994): 24-25.

[Brando 96] Brando, T. "Comparing CORBA & DCE." *Object Magazine 6,* 1 (March 1996): 52-7.

[Brown 96] Brown, A. & Wallnau, K. "A Framework for Evaluating Software Technology." *IEEE Software* (September 1996): 39-49.

[CORBA 96] *Common Object Request Broker: Architecture and Specification* Revision 2.0 [online]. Available WWW <URL: http://www.omg.org> (1996). Also available from the Object Management Group, 492 Old Connecticut Path, Framingham, MA 01701.

[CORBANet 96] Distributed Software Technology Center Home Page [online]. Available WWW <URL: http://corbanet.dstc.edu.au> (1996).

[Deng 95] Deng, R.H., et al. "Integrating Security in CORBA-Based Object Architectures," 50-61. *Proceedings of the 1995 IEEE Symposium on Security and Privacy.* Oakland, CA, May 8-10, 1995. Los Alamitos, CA: IEEE Computer Society Press, 1995.

[Foody 96] Foody, M.A. "OLE and COM vs. CORBA." *Unix Review 14,* 4 (April 1996): 43-5.

[Jell 95] Jell, T. & Stal, M. "Comparing, Contrasting, and Interweaving CORBA and OLE," 140-144. *Object Expo Europe 1995.* London, UK, September 25-29, 1995. Newdigate, UK: SIGS Conferences, 1995.

[Kain 94] Kain, J.B. "An Overview of OMG's CORBA," 131-134. *Proceedings of OBJECT EXPO '94.* New York, NY, June 6-10, 1994. New York, NY: SIGS Publications, 1994.

[Mowbray 93] Mowbray, T.J. & Brando, T. "Interoperability and CORBA-Based Open Systems." *Object Magazine 3,* 3 (September/October 1993): 50-4.

[OMG 96] Object Management Group home page [online]. Available WWW <URL: http://www.omg.org> (1996).

[Roy 95] Roy, Mark & Ewald, Alan. "Distributed Object Interoperability." *Object Magazine 5,* 1 (March/April 1995): 18.

[Steinke 95] Steinke, Steve. "Middleware Meets the Network." *LAN: The Network Solutions Magazine 10,* 13 (December 1995): 56.

[Tibbets 95] Tibbets, Fred. "CORBA: A Common Touch for Distributed Applications." *Data Comm Magazine 24,* 7 (May 1995): 71-75.

[Wallace 96] Wallnau, Kurt & Wallace, Evan. "A Situated Evaluation of the Object Management Group's (OMG) Object Management Architecture (OMA)," 168-178. *Proceedings of the OOPSLA'96.* San Jose, CA, October 6-10, 1996. New York, NY: ACM, 1996. Presentation available [online] FTP. <URL: ftp://ftp.sei.cmu.edu/pub/corba/OOPSLA/present> (1996).

[Watson 96] Watson, A. "The OMG After CORBA 2." *Object Magazine 6,* 1 (March 1996): 58-60.

Author Kurt Wallnau, SEI
kcw@sei.cmu.edu

External Reviewer(s) Dave Carney, SEI
Ed Morris, SEI

Last Modified 10 Jan 97

Component-Based Software Development/ COTS Integration

ADVANCED

Purpose and Origin

Component-based software development (CBSD) focuses on building large software systems by integrating previously-existing software components. By enhancing the *flexibility* and *maintainability* of systems, this approach can potentially be used to reduce software development costs, assemble systems rapidly, and reduce the spiraling maintenance burden associated with the support and upgrade of large systems. At the foundation of this approach is the assumption that certain parts of large software systems reappear with sufficient regularity that common parts should be written once, rather than many times, and that common systems should be assembled through reuse rather than rewritten over and over. CBSD embodies the "buy, don't build" philosophy espoused by Fred Brooks [Brooks 87]. CBSD is also referred to as component-based software engineering (CBSE) [Brown 96a, Brown 96b].

Component-based systems encompass both commercial-off-the-shelf (COTS) products and components acquired through other means, such as nondevelopmental items (NDIs).[1] Developing component-based systems is becoming feasible due to the following:

- the increase in the quality and variety of COTS products
- economic pressures to reduce system development and maintenance costs
- the emergence of component integration technology (see Object Request Broker, pg. 291)
- the increasing amount of existing software in organizations that can be reused in new systems

CBSD shifts the development emphasis from programming software to composing software systems [Clements 95].

Technical Detail

In CBSD, the notion of building a system by writing code has been replaced with building a system by assembling and integrating existing software components. In contrast to traditional development, where system integration is often the tail end of an implementation effort, component integration is the centerpiece of the approach; thus, implementation has given way to integration as the focus of system construction. Because of this, integrability is a key consideration in the decision whether to acquire, reuse, or build the components.

---

1. See the definition of NDI in COTS and Open Systems, pg. 135.

As shown in Figure 4, four major activities characterize the component-based development approach; these have been adapted from Brown [Brown 96b]:

- component qualification (sometimes referred to as suitability testing)
- component adaptation
- assembling components into systems
- system evolution

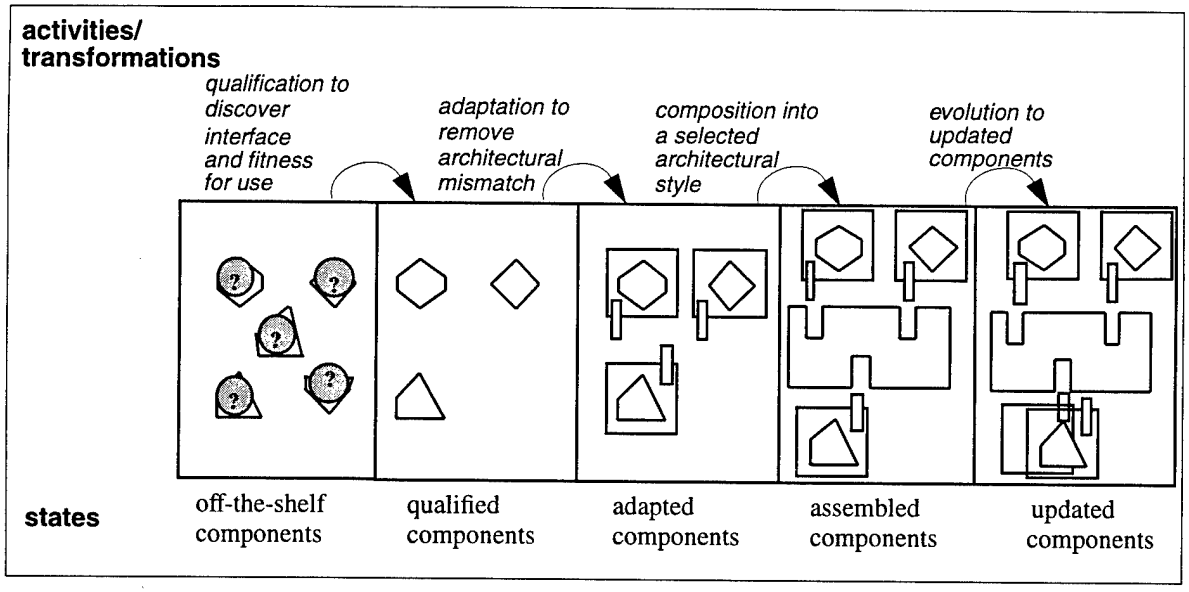


Figure 4: Activities of the Component-Based Development Approach

Each activity is discussed in more detail in the following paragraphs.

Component qualification. Component qualification is a process of determining "fitness for use" of previously-developed components that are being applied in a new system context. Component qualification is also a process for selecting components when a marketplace of competing products exists. Qualification of a component can also extend to include qualification of the development process used to create and maintain it (for example, ensuring algorithms have been validated, and that rigorous code inspections have taken place). This is most obvious in safety-critical applications, but can also reduce some of the attraction of using preexisting components.

There are two phases of component qualification: *discovery* and *evaluation*. In the discovery phase, the properties of a component are identified. Such priorities include component functionality (what services are provided) and other aspects of a component's interface (such as the use of

standards). These properties also include quality aspects that are more difficult to isolate, such as component reliability, predictability, and usability. In some circumstances, it is also reasonable to discover "non-technical" component properties, such as the vendor's market share, past business performance, and process maturity of the component developer's organization. Discovery is a difficult and ill-defined process, with much of the needed information being difficult to quantify and, in some cases, difficult to obtain.

There are some relatively mature evaluation techniques for selecting from among a group of peer products. For example, the International Standards Organization (ISO) describes general criteria for product evaluation [ISO 91] while others describe techniques that take into account the needs of particular application domains [IEEE 93, Poston 92]. These evaluation approaches typically involve a combination of paper-based studies of the components, discussion with other users of those components, and hands-on benchmarking and prototyping.

One recent trend is toward a "product-line" approach that is based on a reusable set of components that appear in a range of software products. This approach assumes that similar systems (e.g., most radar systems) have a similar software architecture and that a majority of the required functionality is the same from one product to the next. (See Domain Engineering and Domain Analysis, pg. 173, for further details on techniques to help determine similarity). The common functionality can therefore be provided by the same set of components, thus simplifying the development and maintenance life cycle. Results of implementing this approach can be seen in two different efforts [Lettes 96, STARSSCAI 95].

Component adaptation. Because individual components are written to meet different requirements, and are based on differing assumptions about their context, components often must be adapted when used in a new system. Components must be adapted based on rules that ensure conflicts among components are minimized. The degree to which a component's internal structure is accessible suggests different approaches to adaptation [Valetto 95]:

- *white box,* where access to source code allows a component to be significantly rewritten to operate with other components
- *grey box,* where source code of a component is not modified but the component provides its own extension language or application programming interface (API) (see pg. 79)
- *black box,* where only a binary executable form of the component is available and there is no extension language or API

Each of these adaptation approaches has its own positives and negatives; however, white box approaches, because they modify source code, can result in serious maintenance and evolution concerns in the long term. Wrapping, bridging, and mediating are specific programming techniques used to adapt grey- and black-box components.

Assembling components into systems. Components must be integrated through some well-defined infrastructure. This infrastructure provides the binding that forms a system from the disparate components. For example, in developing systems from COTS components, several architectural styles are possible:

- *database*, in which centralized control of all operational data is the key to all information sharing among components in the system
- *blackboard*, in which data sharing among components is opportunistic, involving reduced levels of system overhead
- *message bus*, in which components have separate data stores coordinated through messages announcing changes among components
- *object request broker (ORB) mediated*, in which the ORB technology (see pg. 291) provides mechanisms for language-independent interface definition and object location and activation

Each style has its own particular strengths and weaknesses. Currently, most active research and product development is taking place in object request brokers (ORBs) conforming to the Common Object Request Broker Architecture (CORBA) (see pg. 107).[1]

System evolution. At first glance, component-based systems may seem relatively easy to evolve and upgrade since components are the unit of change. To repair an error, an updated component is swapped for its defective equivalent, treating components as plug-replaceable units. Similarly, when additional functionality is required, it is embodied in a new component that is added to the system.

However, this is a highly simplistic (and optimistic) view of system evolution. Replacement of one component with another is often a time-consuming and arduous task since the new component will never be identical to its predecessor and must be thoroughly tested, both in isolation and in combination with the rest of the system. Wrappers must typi-

1. From Wallnau, K. & Wallace, E. *A Robust Evaluation of the Object Management Architecture: A Focused Case Study in Legacy Systems Migration.* Submitted for publication to OOPLSA'96.

cally be rewritten, and side-effects from changes must be found and assessed. One possible approach to remedying this problem is Simplex (see pg. 345).

Usage Considerations

Several items need to be considered when implementing component-based systems:

Short-term considerations

- *Development process.* An organization's software development process and philosophy may need to change. System integration can no longer be at the end of the implementation phase, but must be planned early and be continually managed throughout the development process. It is also recommended that as tradeoffs are being made among components during the development process, the rationale used in making the tradeoff decisions should be recorded and then evaluated in the final product [Brown 96b].
- *Planning.* Many of the problems encountered when integrating COTS components cannot be determined before integration begins. Thus, estimating development schedules and resource requirements is extremely difficult [Vigder 96].
- *Requirements.* When using a preexisting component, the component has been written to a preexisting, and possibly unknown, set of requirements. In the best case, these requirements will be very general, and the system to be built will have requirements that either conform or can be made to conform to the preexisting general requirements. In the worst case, the component will have been written to requirements that conflict in some critical manner with those of the new system, and the system designer must choose whether using the existing component is viable at all.
- *Architecture.* The selection of standards and components needs to have a sound architectural foundation, as this becomes the foundation for system evolution. This is especially important when migrating from a legacy system to a component-based system.
- *Standards.* If an organization chooses to use the component-based system development approach and it also has the goal of making a system open, then interface standards need to come into play as criteria for component qualification. The degree to which a software component meets certain standards can greatly influence the interoperability and portability of a system. Reference the open systems description (see pg. 135) for further discussion.
- *Reuse of existing components.* Component-based system development spotlights reusable components. However, even though organizations have increasing amounts of existing software that can be reused, most often some amount of reengineering must

be accomplished on those components before they can be adapted to new systems.

- *Component qualification.* While there are several efforts focusing on component qualification, there is little agreement on which quality attributes or measures of a component are critical to its use in a component-based system. A useful work that begins to address this issue is "SAAM: A Method for Analyzing the Properties of Software Architecture" [Abowd 94]. Another technique addresses the complexity of component selection and provides a decision framework that supports multi-variable component selection analysis [Kontio 96]. Other approaches, such as the qualification process defined by the US Air Force PRISM program, emphasize "fitness for use" within specific application domains, as well as the primacy of integrability of components [PRISM 96]. Another effort is Product Line Asset Support [CARDS 96].

Long-term considerations

- *External dependencies/vendor-driven upgrade problem.* An organization loses a certain amount of autonomy and acquires additional dependencies when integrating COTS components. COTS component producers frequently upgrade their components based on error reports, perceived market needs and competition, and product aesthetics. DoD systems typically change at a much slower rate and have very long lifetimes. An organization must juggle its new functionality requirements to accommodate the direction in which a COTS product may be going. New component releases require a decision from the component-based system developer/integrator on whether to include the new component in the system. To answer "yes" implies facing an undetermined amount of rewriting of wrapper code and system testing. To answer "no" implies relying on older versions of components that may be behind the current state-of-the-art and may not be adequately supported by the COTS supplier. This is why the component-based system approach is sometimes considered a risk transfer and not a risk reduction approach.
- *System evolution/technology insertion.* System evolution is not a simple plug-and-play approach. Replacing one component often has rippling affects throughout the system, especially when many of the components in the system are black box components; the system's integrator does not know the details of how a component is built or will react in an interdependent environment. Further complicating the situation is that new versions of a component often require enhanced versions of other components, or in some cases may be incompatible with existing components.

  Over the long-term life of a system, additional challenges arise, including inserting COTS components that correspond to new functionality (for example, changing to a completely new communications approach) and "consolidation engineering" wherein several compo-

nents may be replaced by one "integrated" component. In such situations, maintaining external interface compatibility is very important, but internal data flows that previously existed must also be analyzed to determine if they are still needed.

Maturity

To date, the commercial components available and reliable enough for operational systems, and whose interfaces are well-enough understood, have primarily been operating systems, databases, email and messaging systems, office automation software (e.g., calendars, word processors, spreadsheets), and GUI builders (see pg. 205). The number of available components continues to grow and quality and applicability continue to improve. As such, most successful applications have been in the AIS/MIS and C3I areas, with rather limited success in applications having real-time performance, safety, and security requirements. Indeed, in spite of the possible savings, using COTS components to build safety-critical systems where reliability, availability, predictability, and security are essential is frequently too risky [Brown 96b]. An organization will typically not have complete understanding or control of the COTS components and their development.

Examples of apparently successful integration of COTS into operational systems include the following

- Deep Space Network Program at the NASA Jet Propulsion Laboratory [NASA 96a]
- Lewis Mission at NASA's Goddard Space Center [NASA 96b]
- Boeing's new 777 aircraft with 4 million lines of COTS software [Vidger 96]
- Air Force Space and Missile System Center's telemetry, tracking, and control (TT&C) system called the Center for Research Support (CERES) [Monfort 96]

In addition to the increasing availability of components applicable to certain domains, understanding of the issues and technologies required to expand CBSD practice is also growing, although significant work remains. Various new technical developments and products, including CORBA (see pg. 107) and OLE/COM (see pg. 271) [Vidger 96] and changes in acquisition and business practices should further stimulate the move to CBSD.

Costs and Limitations

It is widely assumed that the component-based software development approach, particularly in the sense of using COTS components, will be significantly less costly (i.e., shorter development cycles and lower development costs) than the traditional method of building systems "from scratch." In the case of using such components as databases and oper-

ating systems, this is almost certainly true. However, there is little data available concerning the relative costs of using the component-based approach and, as indicated in Usage Considerations, there are a number of new issues that must be considered.

In addition, if integrating COTS components, an additional system development and maintenance cost will be to negotiate, manage, and track licenses to ensure uninterrupted operation of the system. For example, a license expiring in the middle of a mission might have disastrous consequences.

Dependencies

Adapting preexisting components to a system requires techniques such as API (see pg. 79), wrapping, bridging, or mediating, as well as an increased understanding of architectural interactions and components' properties.

Alternatives

The alternatives include using preexisting components or creating the entire system as a new item.

Complementary Technologies

The advantages of using the CBSD/COTS integration approach can be greatly enhanced by coupling the approach with open systems (see pg. 135).

Domain Engineering and Domain Analysis, pg. 173, aid in identifying common functions and data among a domain of systems which in turn identifies possible reusable components.

Index Categories

Name of technology	Component-Based Software Development/ COTS Integration
Application category	System Allocation (AP.1.2.1), Select or Develop Algorithms (AP.1.3.4), Plan and Perform Integration (AP.1.4.4), Reengineering (AP.1.9.5)
Quality measures category	Maintainability (QM.3.1)
Computing reviews category	Software Engineering Design (D.2.10), Software Engineering Miscellaneous (D.2.m)

References and Information Sources

[Abowd 94] Abowd, G., et al. "SAAM: A Method for Analyzing the Properties of Software Architecture," 81-90. *Proceedings of the 16th International Conference on Software Engineering.* Sorrento, Italy, May 16-21, 1994. Los Alamitos, CA: IEEE Computer Society Press, 1994.

[Brooks 87] Brooks, F. P. Jr. "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer 20,* 4 (April 1987): 10-9.

[Brown 96a] Brown, Alan W. "Preface: Foundations for Component-Based Software Engineering," vii-x. *Component-Based Software Engineering: Selected Papers from the Software Engineering Institute.* Los Alamitos, CA: IEEE Computer Society Press, 1996.

[Brown 96b] Brown, Alan W. & Wallnau, Kurt C. "Engineering of Component-Based Systems," 7-15. *Component-Based Software Engineering: Selected Papers from the Software Engineering Institute.* Los Alamitos, CA: IEEE Computer Society Press, 1996.

[CARDS 96] CARDS [online]. Available WWW <URL: http://www.cards.com/PLAS> (1996).

[Clements 95] Clements, Paul C. "From Subroutines to Subsystems: Component-Based Software Development," 3-6. *Component-Based Software Engineering: Selected Papers from the Software Engineering Institute.* Los Alamitos, CA: IEEE Computer Society Press, 1996.

[IEEE 93] *IEEE Recommended Practice on the Selection and Evaluation of CASE Tools* (IEEE Std. 1209-1992). New York, NY: Institute of Electrical and Electronics Engineers, 1993.

[ISO 91] *Information Technology - Software Product Evaluation - Quality Characteristics and Guidelines for their Use.* Geneve, Switzerland: International Standards Organization/International Electrochemical Commission, 1991.

[Kontio 96] Kontio, J. "A Case Study in Applying a Systematic Method for COTS Selection," 201-209. *Proceedings of the 18th International Conference on Software Engineering.* Berlin, Germany, March 25-30, 1996. Los Alamitos, CA: IEEE Computer Society Press, 1996.

[Lettes 96] Lettes, Judith A. & Wilson, John. *Army STARS Demonstration Project Experience Report* (STARS-VC-A011/003/02). Manassas, VA: Loral Defense Systems-East, 1996.

[Monfort 96] Monfort, Lt. Col. Ralph D. "Lessons Learned in the Development and Integration of a COTS-Based Satellite TT&C System." *33rd Space Congress.* Cocoa Beach, FL, April 23-26, 1996.

[NASA 96a] *COTS Based Development* [online]. Available WWW <URL: http://www-isds.jpl.nasa.gov/isds/cwo's/cwo_23/pbd.htm> (1996).

[NASA 96b] *Create Mechanisms/Incentives for Reuse and COTS Use* [online]. Available WWW <URL:http://bolero.gsfc.nasa.gov/c600/workshops/sswssp4b.htm> (1996).

[Poston 92] Poston R.M. & Sexton M.P. "Evaluating and Selecting Testing Tools." *IEEE Software 9,* 3 (May 1992): 33-42.

[PRISM 96] Portable, Reusable, Integrated Software Modules (PRISM) Program [online]. Available WWW <URL: http://www.cards.com/PRISM/prism_ov.html>(1996).

[STARSSCAI 95] Air Force/STARS Demonstration Project Home Page [online]. Available WWW <URL: http://www.asset.com/stars/afdemo/home/html> (1995).

[Thomas 92] Thomas, I. & Nejmeh. B. "Definitions of Tool Integration for Environments." *IEEE Software 9,* 3 (March 1992): 29-35.

[Valetto 95] Valetto, G. & Kaiser, G.E. "Enveloping Sophisticated Tools into Computer-Aided Software Engineering Environments," 40-48. *Proceedings of 7th IEEE International Workshop on CASE.* Toronto, Ontario, Canada, July 10-14, 1995. Los Alamitos, CA: IEEE Computer Society Press, 1995.

[Vidger 96] Vidger, M.R.; Gentleman, W.M.; & Dean, J. *COTS Software Integration: State-of-the-Art* [online]. Available WWW <http: //wwwsel.iit.nrc.ca/abstracts/NRC39198.abs> (1996).

Authors Capt Gary Haines, AFMC SSSG
ghaines@spacecom.af.mil

David Carney, SEI
djc@sei.cmu.edu

John Foreman, SEI
jtf@sei.cmu.edu

External Reviewer(s) Paul Kogut, Lockheed Martin, Paoli, PA
Ed Morris, SEI
Tricia Oberndorf, SEI
Kurt Wallnau, SEI

Last Modified 10 Jan 97

Computer System Security— an Overview ADVANCED

Purpose and Origin

C4I systems include networks of computers that provide real-time situation data for military decision makers and a means of directing response to a situation. These networks collect data from sensors and subordinate commands. That data is fused with the existing situation status data and presented by the C4I system to decision makers through display devices. C4I networks today may incorporate two general types of networks: networks of Multi-level Secure (MLS) Systems, and Intranets of single level systems. Figure 5 shows the relevant major security components of a C4I computer system network.

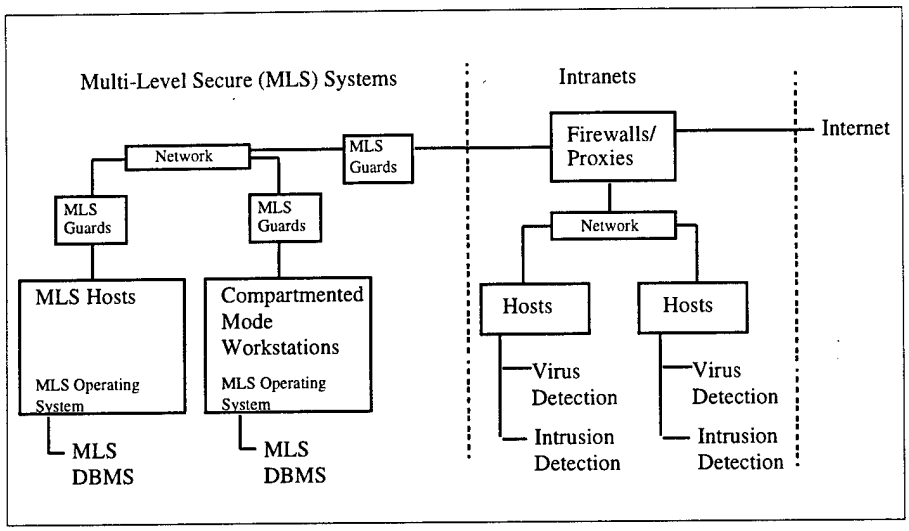


Figure 5: Computer System Security in C4I Systems

This technology description is tutorial in nature. It provides a general overview of key concepts and introduces key technologies. Detailed discussions of the individual technologies can be found in the referenced technology descriptions.

Technical Detail

Some computers in the network are hosts that collect and process data. A host can be a mainframe, a server, a workstation, or a PC. It may perform the function of an application processor, a communication processor, a database processor, a display processor, or a combination. The security mode for the host may be single-level or multi-level. A single-level host processes all data as though it was one security level. A multi-level host can process data at different security levels, identify and isolate data in the appropriate levels or categories, and distribute data only to the appropriately cleared users.

C4I systems benefit from multi-level security implementations because C4I systems fuse data from sources with a wide range of security levels and provide status, warning data, or direction to war fighting systems that may be at lesser security levels. An MLS operating system (see pg. 377) provides the software that makes a host MLS. A particular kind of MLS host is the Compartmented Mode Workstation (CMW). A CMW is a MLS host that has been evaluated to satisfy the Defense Intelligence Agency CMW requirements [Woodward 87] in addition to the Trusted Computer System Evaluation Criteria [DOD 85]. A MLS host may use a MLS DBMS (see pg. 261) to store and retrieve data at multiple security levels. A MLS guard provides a secure interface across a security boundary between systems operating at different security levels or modes.

MLS guards may allow data across the interface automatically or may require manual review of data and approval of transfer on an attached terminal. They also may control data transfer across the interface in both directions or be limited to allowing data to be transferred one way, usually from the low security level side of a security boundary to the high security level side. One-way guards are usually the easiest to implement and accredit for use. Data integrity is an issue with one-way guards because an acknowledgment message can not be used. Recent research in one-way guards has addressed allowing an acknowledgment message.

Intranets use the same kind of networking software (e.g., TCP/IP, Telnet, Netnews, DNS, browsers, home pages) that is used on the Internet, but Intranets use them on a private dedicated network. They are in essence a private Internet. They are used in a growing number of ways in many military and corporate networks including mission performance, off-line processing of raw data, administrative support, and mail networks. They may be incorporated into C4I systems using firewalls or proxies (see pg. 191) and MLS guards. Firewalls or proxies may be used to provide a security interface to the Internet. If the Intranets are to be connected to MLS systems, they must be connected through MLS guards. In an environment with Intranet hosts, a major concern is virus detection (see pg. 387) and intrusion detection (see pg. 217). PCs on a network are particularly susceptible to virus attacks from other hosts on the network or the Internet. PCs are also vulnerable to viruses carried on floppy disks. Since PCs are now in most homes, transfer of files from home to work via floppy disk provides the risk of introducing a virus into the Intranet. PCs are more vulnerable to viruses than UNIX-based workstations or mainframes because the PC has no memory protection hardware and the operating system (DOS and Windows) allows a program to access any part of memory or disk.

Security across the networks in a C4I system is crucial. Traditionally this security is provided by physically protecting the equipment and cables in the network for localized networks. When that is not possible, the network connections are encrypted using encryption hardware in the communications paths. End-to-end encryption is an alternative that encrypts the data using software before it is put on the network and decrypts it after it has been taken off of the network. Then non-encrypted circuits can be used for communications.

Any encryption system involves the distribution of keys used by the encryption algorithm for the encryption/decryption of messages and data. Encryption keys must be replaced periodically to enhance security or when the key has been compromised or lost. Traditionally these keys have been distributed through couriers or encrypted circuits. Public key cryptography provides a means of electronic encryption key distribution that can lower the security risk and administrative workload associated with encryption.

Data integrity is another issue associated with the networks used in C4I systems. Public key digital signatures (see pg. 309) and providing for nonrepudiation in network communications (see pg. 269) are two means to enhance data integrity. Public key digital signatures, which make use of public key encryption and message authentication codes, are a means to authenticate that data came from the person identified as the sender and that the data has not been modified. The nonrepudiation process uses a digital signature and a trusted arbitrator process to assure that a particular message has been sent and received and to establish the time when this occurred.

Usage Considerations

MLS systems require specialized knowledge to build, accredit, and maintain. The cost of MLS systems can be high. The system development overhead and operational performance overhead associated with MLS systems are substantial. They are difficult to implement in an "open" configuration because open requirements sometimes conflict with MLS requirements. On the other hand, using MLS techniques may be the only allowable way to construct some C4I systems. Operational security vulnerabilities may be unacceptable without MLS implementations. Procedural security approaches may be too slow for an operational C4I system as a non-MLS approach. A single-level system approach may be too restrictive. For example, a secret single-level system that contains unclassified, confidential, and secret data will not release confidential data to a user who is cleared for confidential and needs the data. That is because the system cannot determine what data is confidential rather than secret.

Further usage discussions are addressed in individual technology descriptions.

The National Security Agency (NSA) Multilevel Information Systems Security Initiative (MISSI) is an evolutionary effort intended to provide better MLS capability in a cost-effective manner [MISSI 96]. This effort was initiated after the Gulf War when it was recognized that war fighting commanders needed MLS systems in order to incorporate intelligence and other highly classified data into their planning and operations in a timely manner. The MISSI effort is developing a set of building block products that can be obtained commercially to construct an MLS system. The initial products include the FORTEZZA crypto cards and associated FORTEZZA ready workstation applications to control access to and protect data on a workstation in a network environment. Other products include high-assurance guards and firewalls to provide access control and encryption services between the local security boundary and external networks. MISSI will also include secure computing products that provide high-trust operating systems and application programs for MLS hosts, and network encryption and security management products. These products can be incorporated into developing MLS systems as the products become available.

Maturity

See individual technologies.

Costs and Limitations

See individual technologies.

Index Categories

Name of technology	Computer System Security— an Overview
Application category	Information Security (AP.2.4)
Quality measures category	Security (QM.2.1.5)
Computing reviews category	Operating Systems Security & Protection (D.4.6), Security & Protection (K.6.5), Computer-Communications Networks Security and Protection (C.2.0)

References and Information Sources

[Abrams 95] Abrams, Marshall D.; Jajodia, Sushil; & Podell, Harold J. *Information Security An Integrated Collection of Essays*. Los Alamitos, CA: IEEE Computer Society Press, 1995.

[Woodward 87] Woodward, John. *Security Requirements for High and Compartmented Mode Workstations* (MTR 9992, DDS 2600-5502-87). Washington, DC: Defense Intelligence Agency, 1987.

[DoD 85] *Department of Defense (DoD) Trusted Computer System Evaluation Criteria (TCSEC)* (DoD 5200.28-STD 1985). Fort Meade, MD: Department of Defense, 1985.

[MISSI 96] MISSI Web site [online]. Available WWW <URL: http://Beta.MISSILAB.COM:9000/> (1996).

[Russel 91] Russel, Deborah & Gangemi, G.T. Sr. *Computer Security Basics*. Sebastopol, CA: O'Reilly & Associates, Inc., 1991.

[White 96] White, Gregory B.; Fisch, Eric A.; & Pooch, Udo W. *Computer System and Network Security*. Boca Raton, FL: CRC Press, 1996.

Author Tom Mills, Lockheed Martin
TMILLS@ccs.lmco.com

External Reviewer(s) Brian Gallagher, SEI

Last Modified 10 Jan 97

COTS and Open Systems

ADVANCED

Purpose and Origin

One of the latest trends in systems development is to make greater use of commercial-off-the-shelf (COTS) products. While this change has been encouraged for many years for all kinds of systems development, especially in the Department of Defense (DoD), it is only in the early 1990s that the practice has been mandated by everyone from industry executives to Congress.

At the same time, an open systems approach to develop systems has been gaining popularity, with visions of open systems that are "plug-and-play" compatible, where components from one supplier can be easily replaced by those from another supplier. Advocates of open systems often confuse them with the use of COTS products, making it difficult for the average engineer to know just what (s)he should be doing to develop (and maintain) systems more effectively.

These two concepts— the use of COTS products and the creation of open systems— are closely related and complementary, although definitely *not* synonymous. The purpose of this technology description is to

- define/clarify what each is
- explain the differences between them
- examine the benefits each brings to the development, maintenance, and evolution of systems

As a brief summary

- COTS products hold the potential for cost-effective acquisition of components and advancing technology, but they are not necessarily open.
- Open systems emphasize (1) the use of interface standards and (2) the use of implementations that conform to those standard interfaces. Open systems provide stability and a framework for the effective use of COTS products and other non-developmental items (NDI) (see pg. 137) through the use of interface standards. *Well-chosen interface standards can out last any particular product, vendor, or technology.*
- It is possible to use COTS products without creating an open system; similarly, it is possible to create an open system without significant use of COTS products or NDI.
- COTS products and an open systems approach are both means to important system goals of improving the quality and performance of our systems, developing them more quickly, and sustaining them more cost-effectively. The greatest advantage can be gained from using these two approaches together.

For further detail on COTS, open systems, and component-based software development approaches, see Component-Based Software Development/ COTS Integration, pg. 119.

Technical Detail

COTS. The term "COTS" is meant to refer to things that one can buy, *ready-made,* from some manufacturer's virtual store shelf (e.g., through a catalogue or from a price list). It carries with it a sense of getting, at a reasonable cost, something that already does the job. It replaces the nightmares of developing unique system components with the promises of fast, efficient acquisition of cheap (or at least cheaper) component implementations.

Because of the need for precision in procurement, the federal government has defined the term "commercial item." The full text of this definition can be found in the Federal Acquisition Regulations (FARs); the following is a summary [FAR 93]:

A *commercial item* is

1. property[1] customarily used by the general public in the course of normal business and
   - has been sold, leased, or licensed (or offered for sale, lease or license) to the public, or
   - is not yet available in the commercial marketplace but will be available within a reasonable period
2. any item that would satisfy (1) but for modifications customarily available in the commercial marketplace or minor modifications made to meet Federal Government requirements
3. services for installation, maintenance, repair, training, etc. if such services are procured for support of an item in (1) or (2) above, as offered to the public

In recognition of the desirability of staying close to the commercial technology base even where government requirements diverge, the FARs also include this related definition:

Commercial-type product means a commercial product (a) modified to meet some Government-peculiar physical requirement or addition, or (b) otherwise identified differently from its normal commercial counterparts [FAR 93].

1. "Property" in this definition explicitly excludes real property.

Most people would agree that these ideas approximate the meaning of "commercial-off the-shelf" (COTS) products. The salient characteristics of a COTS product are the following:

- it exists *a priori*
- it is available to the general public
- it can be bought (or leased or licensed)

Non-developmental item. A closely-related term that is often heard in government (especially DoD) circles is "nondevelopmental item" (NDI). A summary definition of a *nondevelopmental item* is [FAR 93]:

1. any commercial item
2. any previously developed item in use by a U.S. agency (federal, state, or local) or a foreign government that has a mutual defense agreement with the U.S.
3. any item described in (1) or (2) above that requires only minor modification normally available in the commercial marketplace to meet requirements
4. any item currently being produced that does not meet (1), (2), or (3) above only because it is not yet in use or is not yet available in the commercial marketplace.

The key point here is that NDI refers to something that was developed by someone else. It might have been developed by a commercial interest, but typically it will have been for some other government, department, or agency. A large-scale example of an NDI would be a fighter aircraft that was developed by some other nation. A more meaningful example in the current context would be a radar developed for one aircraft that is available for use in another aircraft. The salient characteristics of a non-developmental item are the following:

- it exists, although not necessarily off some vendor's "shelf"
- it is available, although not necessarily to the general public
- it can be obtained for use, although perhaps not by purchase or lease

While there are certain reasons for using caution in applying the definitions of COTS and NDI (e.g., how safe is a "minor modification," and what if it just *looks* like a vendor has a product, whereas it is in reality just vaporware?), they do fairly characterize the features that are of interest to those who believe that "buying COTS" is desirable and beneficial. How-

ever, although closely related, there are differences between NDI and COTS items:

- COTS products would most likely be found in some sort of catalogue or price list, whereas it may be more difficult to discover the existence of NDI.
- The range of possibilities opened up by NDI is broader than what COTS products alone can offer, but NDI could lack the commercial leverage that is sought in the use of COTS products.

Open systems. The basic tenet of open systems is the use of interface standards in the engineering and design of systems, coupled with the use of implementations (preferably, but not necessarily, COTS and non-developmental items (NDI)) that conform to those interface standards. This provides a stable basis on which to make decisions about the system, particularly with regard to its evolution. An open systems approach has the potential to reduce the developmental cost and schedule of systems while maintaining or even improving performance. The dependence on stable interface standards makes open systems more adaptable to advances in technology and changes in the marketplace.

When people use the phrase open systems, they most often have in mind a system that is flexible and adaptive, one that is "open" to inclusion of many products from many sources. The phrase open systems often carries with it an image of easy "plug-and-play" between components and products that were *not* necessarily originally designed to work together. Open systems also hold out the promise of being able to take immediate advantage of new technology as it emerges, because it should be easier to plug in new technology, either in place of an old component(s) or as a new extension of the system.

Many different definitions of open system have been offered in the past. To find a truly workable one, we must look more closely at what it takes to make this vision a reality. For the purposes of this technology description, open systems is defined as follows [Myers 96]:

An *open system* is

> a collection of interacting software, hardware, and human components, designed to satisfy stated needs, with the interface specification of components
>
> - fully defined
> - available to the public
> - maintained according to group consensus, and

in which the implementations of components are conformant to the specification.

One key part of the definition addresses a set of criteria for the interface specifications/standards. Not only must they be fully defined, but they must also be available to the public. This implies that cost and public access may not be prohibitive constraining factors; that is, the specification cannot be available only to a selected group of people who have some special interest. Anyone is free to obtain a copy of the specification (perhaps at the cost of duplication and distribution, perhaps even at the cost of a small license fee) and they are also free to produce and sell implementations of that specification. It is also very important that the specification is of interest to a wide range of parties and is not exclusively under the control of any single vendor. To this end, the definition includes the idea that maintenance of the specification is by group consensus. Taken together, these criteria come very close to requiring that the interface specification be a "standard."

The main benefit of this definition of open system is that it is *operational.* That is, it can be applied to a single system at a given point in time. In contrast, most other popular definitions identify desirable system qualities that open systems are expected to display, such as portability, interoperability, and scalability. Unfortunately, there is no way to measure a system with respect to these qualities at a single point in time (e.g., "Portable" to what platforms? And how many? "Interoperable" with what other systems or components? And how many? "Scalable" for what use? To what size?). Each of these qualities implies a relationship, either between the subject system and some other unspecified one(s) or between the subject system and itself over time.

This definition also supports the vision of what people hope to achieve with open systems. The very phrase "plug-and-play" brings to mind children's toys like Tinker Toys™ and Legos™. The key to them is a small set of well-defined, consistently-enforced interfaces. It also invokes the images of hardware components that can be plugged together because, for example, the pins and configuration of the female connector are perfectly complementary to those of the male connector. All these schemes have interface standards in common.

Most of the interface standards used in computer-based systems are for application program interfaces (APIs), data formats, or protocols. For all of these kinds of interface standards, one can find fully-defined specifications; without such clear definition in the specifications, wide variation quickly emerges among implementations, and this undermines the in-

tended compatibility. Interface standards are made widely available to the public to generate a thriving market for components that can be plugged together. They are maintained using many forms of group consensus; this precludes one vendor or group from making arbitrary changes to the interface standard that will limit competition and availability of alternative products.

Finally, for many of these interface standards it is possible to tell whether or not a given implementation really matches the specification; this is called *conformance*. If the implementations all match the specification/standard closely enough, then one kind of incompatibility between components can be reduced if not eliminated, and it *may* be possible to "plug" them into a system and get them to "play" with the other components.[1] On the other hand, if implementations only loosely implement the standard or if incompatible interpretations cannot be detected before trying to integrate a component into the system, then it is less likely that the envisioned flexibility and adaptability can be realized.

It is important to realize that it is possible to create an open system, based on interface standards, in which no COTS products or NDI are used. This might be necessary in a situation where, for example, no COTS product conforming to the interface standard also meet other system requirements, such as for real-time performance or security. Although one would not gain the economic and schedule advantages of using a component implementation that already existed and was shared and supported by a number of users, the interface standards would still provide the framework for future evolution of the system (provided vendors do eventually pick up the standard and produce conformant products). Potentially some future product may emerge that does meet all the requirements. In the mean time, the system enjoys the clarity and stability of a well-defined specification.

1. It should be noted that interface specifications are in general not sufficient to ensure full "plug-and-play" operation. In practice, the real interface between two components of a system consists of all the assumptions that each makes about the other. APIs, data formats, and protocols address a large number of these assumptions, but by no means all of them. It remains for further investigations to determine the full set of interface knowledge that must be standardized to ever get really close to an ideal "plug-and-play" system creation process.

Usage Considerations

There currently is a very strong push within the federal government, particularly DoD, to make more use of COTS products and NDI.[1] In addition to action by DoD leaders, the Federal Acquisition Streamlining Acts of 1994 and 1995 directed the increased use of commercial items, coupled with several adjustments to the federal procurement regulations to encourage the new approach.

The reasoning behind these directives and laws is that government organizations typically spend far too much effort on defining to the lowest level of detail the desired characteristics of systems and how the contractors are to build those systems to achieve those characteristics. Thus a lot of resources are expended developing systems and components that often already exist— or exist in "good enough" form with nearly the same capabilities— elsewhere. The prevailing, and time-consuming, approach is to always develop from the ground up; this approach results in unique systems each time. The result is systems that are

- very expensive, with only one customer to bear the development and maintenance costs over the life of the component or system
- inflexible and unable to easily capitalize on advances in technology
- historically fielding technology that is in excess of ten years old

Shifting to a paradigm in which systems are built primarily of components that are available commercially offers the opportunity to lower costs by sharing them with other users, thus amortizing them over a larger population, while taking advantage of the investments that industry is putting into the development of new technologies.

Open systems can have a positive impact on either new systems development or in the context of legacy systems. Although there is generally more decision-making freedom in the case of a new development, open systems can nevertheless help shape an evolutionary path for a legacy system that will help turn it into a more flexible and maintainable system.

Many initiatives are under way, both in the DoD and in individual services, agencies, and companies, that are designed to promote the use of an open systems approach and to secure even greater benefits than can be realized from the use of COTS products alone. These initiatives are oc-

1. In June 1994 Secretary of Defense William Perry directed that DoD acquisitions should make maximum use of performance specifications and commercial standards. In November 1994 Undersecretary of Defense (Acquisition and Technology) Paul Kaminski directed "that 'open systems' specifications and standards be used for acquisition of weapon systems electronics to the greatest extent practical."

curring because projects have been learning the hard way that "just buying COTS" does not necessarily secure all of the benefits desired. There are other problems and sources of risk introduced by the use of COTS products.

COTS products are not necessarily open. That is, they do not necessarily conform to any recognized interface standards. Thus it is possible (in fact, likely) that using a COTS product commits the user to proprietary interfaces and solutions that are not common with any other product, component, or system. If the sole objective is the ability to capture new technology more cheaply, then the use of COTS products that are not open will do. But when one considers the future of such a system, the disadvantages of this approach become apparent. Many DoD systems have a 30- to 50-year lifetime, while the average COTS component is upgraded every 6 to 12 months and new technology appears on the scene about every 18 to 24 months. Thus any money that is saved by procuring a COTS product with proprietary interfaces will quickly be lost in maintenance as products and interfaces change— the ability to migrate cost-effectively to other products and other technologies in the future will have been lost.

Even if the expected lifetime of a system is only 5 to 10 years, the fluctuations in COTS products and technology result in a state of constant change for any system employing them. Interface standards provide a source of stability in the midst of all this. Without such standards every change in the marketplace can impose an unanticipated and unpredictable change to systems that use products found in the marketplace. This situation is particularly painful when the vendor stops supporting the product or goes out of business altogether, thus forcing a change to a different product or vendor.

Program managers and lead engineers should also know that the depth of understanding and technical and management skills required on a project team are not necessarily diminished or decreased because of the use of COTS or open systems. Arguably, the skills and understanding needed increase because of the potential complexity of integration issues; the need to seriously consider longer term system evolution as part of initial development; and the need to make informed decisions about which products and standards are best.

Paradoxically, given the desire to produce systems more quickly, the emphasis on standards can actually be something of an inhibitor. Some standards efforts, in their desire to achieve maximum consensus, have very long cycle times (five or more years), which certainly do not fit well

with product development and release cycles. This conflict is of concern and is being addressed by some standard groups, but it has led some projects to become involved with consortia-oriented standards efforts and also with defacto industry standards. While these are often practical alternatives, they do have attendant risks; the defacto standards may be proprietary, for example, and this limit long-term evolution.

Maturity

The open systems concept has been at least partially introduced into C3I systems, but it has been difficult to move these ideas into the realm of real-time embedded systems, particularly weapon systems, where it is much more difficult to find standards that meet a system's requirements, for example, in cases of extreme real-time performance or security concerns.

There is limited documented experience with the open systems approach. An example of successful use in the DoD is the Intelligence and Electronics Warfare Common Sensor (IEWCS) program [IEWCS 96]. A survey of the awareness, understanding, and implementation of open system concepts within the DoD has been completed recently by the Open Systems Joint Task Force (OSJTF) [OSJTF 96].

Costs and Limitations

An open systems approach requires investments in the following areas early in a program's life cycle and on an ongoing basis:

- market surveys to determine the availability of standards
- standards selection
- standards profiling— the coordination and tailoring of standards to work together
- selection of standards-compliant implementations

These costs/activities are the necessary foundation for creating systems that serve current needs and yet can grow and advance as technology advances and the marketplace changes.

A separate cost is the continued willingness of the government to invest in standards development and maturation activities. While these activities are most often handled at high government levels concerned with standards development and usage (for example, Defense Information Systems Agency (DISA) in the DoD), it is still important for individual programs (especially the larger programs) to stay informed in this area. For example, individual programs should be concerned about the following issues: When are revisions to specific standards coming out? When are ballots on the revisions going to occur? Where are the implementations headed?

Index Categories

Name of technology	COTS and Open Systems
Application category	Interfaces Design (AP.1.3.3), Software Architecture (AP.2.1)
Quality measures category	Openness (QM.4.1.2), Interoperability (QM.4.1), Maintainability (QM.3.1)
Computing reviews category	Software Engineering Design (D.2.10), Software Engineering Miscellaneous (D.2.m)

References and Information Sources

[Carney 97] Carney, David & Oberndorf, Tricia. *Ten Commandments of COTS*. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997.

[FAR 93] *Federal Acquisition Regulations*. Washington, DC: General Services Administration, 1993.

[IEWCS 96] *Open Systems Joint Task Force Case Study of U.S. Army Intelligence and Electronic Warfare Common Sensor* (IEWCS) [online]. Available WWW <URL: http://www.acq.osd.mil/osjtf/caserpt.htm> (1996).

[Myers 96] Meyers, Craig & Oberndorf, Tricia. *Open Systems: The Promises and the Pitfalls*. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1996.

[OSJTF 96] *Open Systems Joint Task Force Baseline Study* [online]. Available WWW <URL: http://www.acq.osd.mil/osjtf/evidence.html> (1996).

Author

Tricia Oberndorf, SEI
po@sei.cmu.edu

John Foreman, SEI
jtf@sei.cmu.edu

Internal Reviewer John Foreman

Last Modified 10 Jan 97

Cyclomatic Complexity

ADVANCED

Note

We recommend reading Maintenance of Operational Systems— an Overview, pg. 237, before reading this description; it offers a view of the life cycle of software from development through reengineering. We also recommend concurrent reading of Maintainability Index Technique for Measuring Program Maintainability, pg. 231, which illustrates a specific application of cyclomatic complexity to quantify the maintainability of software. These descriptions provide a framework for assessing the applicability of cyclomatic complexity and other technologies to a specific environment.

Purpose and Origin

Cyclomatic complexity is the most widely used member of a class of static software metrics. Cyclomatic complexity may be considered a broad measure of *soundness* and *confidence* for a program. Introduced by Thomas McCabe in 1976, it measures the number of linearly-independent paths through a program module. This measure provides a single ordinal number that can be compared to the complexity of other programs. Cyclomatic complexity is often referred to simply as program complexity, or as McCabe's complexity. It is often used in concert with other software metrics. As one of the more widely-accepted software metrics, it is intended to be independent of language and language format [McCabe 94].

Cyclomatic complexity has also been extended to encompass the design and structural complexity of a system [McCabe 89].

Technical Detail

The cyclomatic complexity of a software module is calculated from a connected graph of the module (that shows the topology of control flow within the program):

Cyclomatic complexity (CC) = E - N + p

where E = the number of edges of the graph

N = the number of nodes of the graph

p = the number of connected components

To actually count these elements requires establishing a counting convention (tools to count cyclomatic complexity contain these conventions). The complexity number is generally considered to provide a stronger measure of a program's structural complexity than is provided by counting lines of code. Figure 6 is a connected graph of a simple program with a cyclomatic complexity of seven. Nodes are the numbered locations, which correspond to logic branch points; edges are the lines between the nodes.

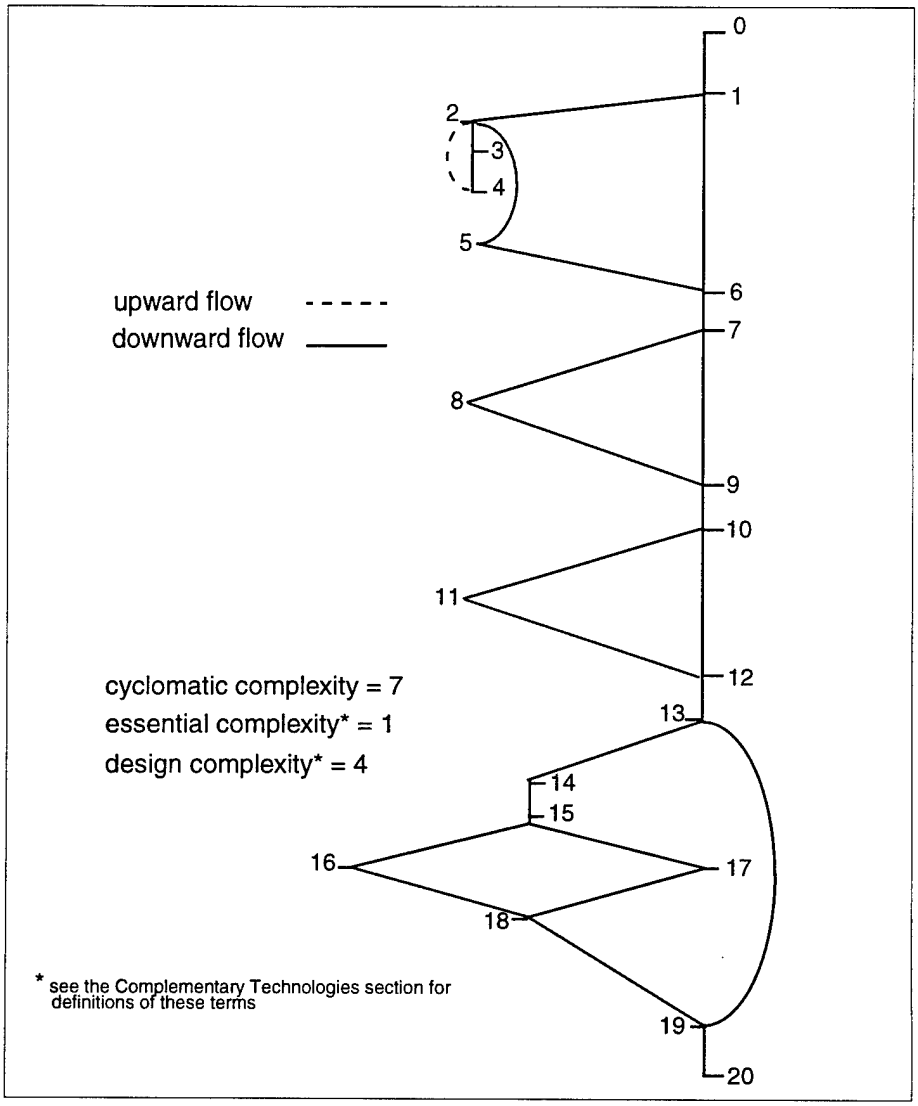


Figure 6: Connected Graph of a Simple Program

A large number of programs have been measured, and ranges of complexity have been established that help the software engineer determine a program's inherent risk and stability. The resulting calibrated measure can be used in development, maintenance, and reengineering situations to develop estimates of risk, cost, or program stability. Studies show a correlation between a program's cyclomatic complexity and its error frequency. A low cyclomatic complexity contributes to a program's *understandability* and indicates it is amenable to *modification* at lower risk than a more complex program. A module's cyclomatic complexity is also a strong indicator of its *testability* (see Test planning under Usage Considerations on pg. 147).

A common application of cyclomatic complexity is to compare it against a set of threshold values. One such threshold set is as follows:

Cyclomatic Complexity	Risk Evaluation
1-10	a simple program, without much risk
11-20	more complex, moderate risk
21-50	complex, high risk program
greater than 50	untestable program (very high risk)

Usage Considerations

Cyclomatic complexity can be applied in several areas, including

- *Code development risk analysis.* While code is under development, it can be measured for complexity to assess inherent risk or risk buildup.
- *Change risk analysis in maintenance.* Code complexity tends to increase as it is maintained over time. By measuring the complexity before and after a proposed change, this buildup can be monitored and used to help decide how to minimize the risk of the change.
- *Test Planning.* Mathematical analysis has shown that cyclomatic complexity gives the exact number of tests needed to test every decision point in a program for each outcome. Thus, the analysis can be used for test planning. An excessively complex module will require a prohibitive number of test steps; that number can be reduced to a practical size by breaking the module into smaller, less-complex sub-modules.
- *Reengineering.* Cyclomatic complexity analysis provides knowledge of the structure of the operational code of a system. The risk involved in reengineering a piece of code is related to its complexity. Therefore, cost and risk analysis can benefit from proper application of such an analysis.

Cyclomatic complexity can be calculated manually for small program suites, but automated tools are preferable for most operational environments. For automated graphing and complexity calculation, the technology is language-sensitive; there must be a front-end source parser for each language, with variants for dialectic differences.

Cyclomatic complexity is usually only moderately sensitive to program change. Other measures (see Complementary Technologies on pg. 148) may be very sensitive. It is common to use several metrics together, either as checks against each other or as part of a calculation set (see Maintainability Index Technique for Measuring Program Maintainability, pg. 231).

Maturity

Cyclomatic complexity measurement, an established but evolving technology, was introduced in 1976. Since that time it has been applied to tens of millions of lines of code in both Department of Defense (DoD) and commercial applications. The resulting base of empirical knowledge has allowed software developers to calibrate measurements of their own software and arrive at some understanding of its complexity. Code graphing and complexity calculation tools are available as part (or as options) of several commercial software environments.

Costs and Limitations

Cyclomatic complexity measurement tools are typically bundled inside commercially-available CASE toolsets. It is usually one of several metrics offered. Application of complexity measurements requires a small amount of training. The fact that a code module has high cyclomatic complexity does not, by itself, mean that it represents excess risk, or that it can or should be redesigned to make it simpler; more must be known about the specific application.

Alternatives

Cyclomatic complexity is one measure of structural complexity. Other metrics bring out other facets of complexity, including both structural and computational complexity, as shown in the following table.

Complexity Measurement	Primary Measure of
Halstead metrics (see pg. 209)	Algorithmic complexity, measured by counting operators and operands
Henry and Kafura metrics	Coupling between modules (parameters, global variables, calls)
Bowles metrics	Module and system complexity; coupling via parameters and global variables
Troy and Zweben metrics	Modularity or coupling; complexity of structure (maximum depth of structure chart); calls-to and called-by
Ligier metrics	Modularity of the structure chart

Marciniak offers a more complete description of complexity measures and the complexity factors they measure [Marciniak 94].

Complementary Technologies

The following three metrics are specialized measures that are used in specific situations:

1. *Essential complexity.* This measures how much unstructured logic exists in a module (e.g., a loop with an exiting GOTO statement).

   The program in Figure 6, pg. 146 has no such unstructured logic, so its essential complexity value is one.

2. *Design complexity.* This measures interaction between decision logic and subroutine or function calls.

   The program in Figure 6, pg. 146 has a design complexity value of 4, which is well within the range of desirability.

3. *Data complexity.* This measures interaction between data references and decision logic.

Other metrics that are "related" to Cyclomatic complexity in general intent are also available in some CASE toolsets.

The metrics listed in Alternatives, pg. 148, are also complementary; each metric highlights a different facet of the source code.

Index Categories

Name of technology	Cyclomatic Complexity
Application category	Test (AP.1.4.3), Reapply Software Lifecyle (AP.1.9.3), Reverse Engineering (AP.1.9.4), Reengineering (AP.1.9.5)
Quality measures category	Maintainability (QM.3.1), Testability (QM.1.4.1), Complexity (QM.3.2.1), Structuredness (QM.3.2.3)
Computing reviews category	Software Engineering Metrics (D.2.8), Complexity Classes (F.1.3), Tradeoffs Among Complexity Measures (F.2.3)

References and Information Sources

[Marciniak 94] Marciniak, John J., ed. *Encyclopedia of Software Engineering,* 131-165. New York, NY: John Wiley & Sons, 1994.

[McCabe 89] McCabe, Thomas J. & Butler, Charles W. "Design Complexity Measurement and Testing." *Communications of the ACM 32,* 12 (December 1989): 1415-1425.

[McCabe 94] McCabe, Thomas J. & Watson, Arthur H. "Software Complexity." *Crosstalk, Journal of Defense Software Engineering 7,* 12 (December 1994): 5-9.

[Perry 88] Perry, William E. *A Structured Approach to Systems Testing.* Wellesley, MA: QED Information Sciences, 1988.

Author

Edmond VanDoren, Kaman Sciences
bvandoren-cos3@kaman.com

Last Modified 10 Jan 97

Database Two Phase Commit

ADVANCED

Note

We recommend Three Tier Software Architectures, pg. 367, as prerequisite reading for this technology description.

Purpose and Origin

Since the 1980s, two phase commit technology has been used to automatically control and monitor commit and/or rollback activities for transactions in a distributed database system. Two phase commit technology is used when data updates need to occur simultaneously at multiple databases within a distributed system. Two phase commits are done to maintain data integrity and *accuracy* within the distributed databases through synchronized locking of all pieces of a transaction. Two phase commit is a proven solution when data integrity in a distributed system is a requirement. Two phase commit technology is used for hotel and airline reservations, stock market transactions, banking applications, and credit card systems. For more details on two phase commit see the *ORACLE7 Server Concept Manual* and *The Performance of Two-Phase Commit Protocols in the Presence of Site Failures* [ORACLE7 92, UCSB 94].

Technical Detail

As shown in Figure 7, applying two phase commit protocols ensures that execution of data transactions are synchronized, either all committed or all rolled back (not committed) to each of the distributed databases.

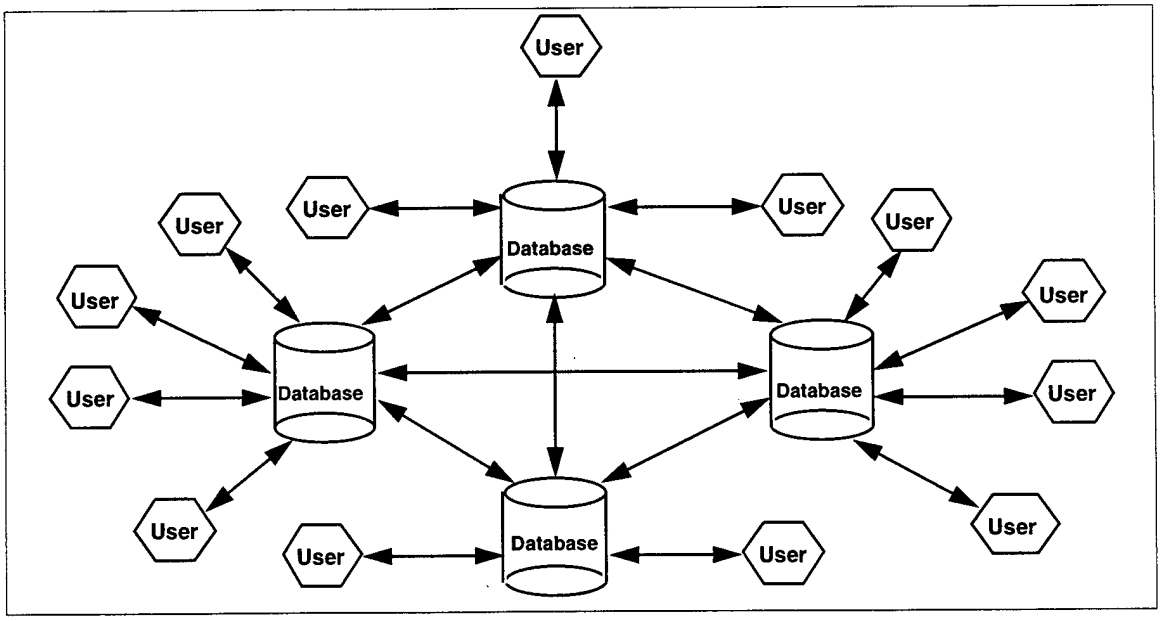


Figure 7: Distributed Databases When Two Phase Commit Happens Simultaneously Through the Network

When dealing with distributed databases, such as in the client/server architecture, distributed transactions need to be coordinated throughout the network to ensure data integrity for the users. Distributed databases using the two phase commit technique update all participating databases simultaneously.

Unlike non-distributed databases (see Figure 8), where a single change is or is not made locally, all participating databases must all commit or all rollback in distributed databases, even if there is a system or network failure at any node. This is how the two phase commit process maintains system data integrity.

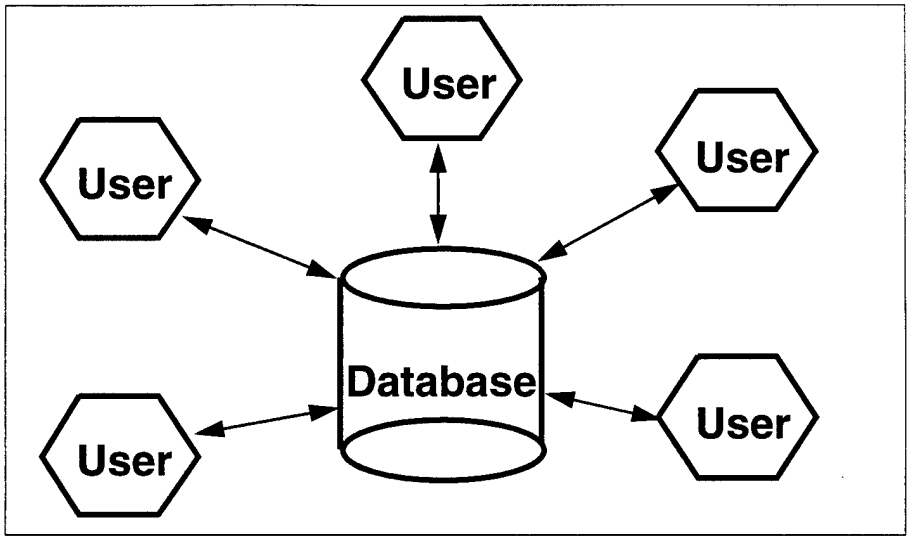


Figure 8: Non-Distributed Databases Make Only Local Updates

Two phase commit has two distinct processes that are accomplished in less than a fraction of a second:

1. The Prepare Phase, where the global coordinator (initiating database) requests that all participants (distributed databases) will promise to commit or rollback the transaction. (Note: Any database could serve as the global coordinator, depending on the transaction.)
2. The Commit Phase, where all participants respond to the coordinator that they are prepared, then the coordinator asks all nodes to commit the transaction. If all participants cannot prepare or there is a system component failure, the coordinator asks all databases to roll back the transaction.

Should there be a machine, network, or software failure during the two phase commit process, the two phase commit protocols will automatically and transparently complete the recovery with no work from the database administrator. This is done through use of pending transaction

tables in each database where information about distributed transaction is maintained as they proceed through the two phase commit. Information in the pending transaction table is used by the recovery process to resolve any transaction of questionable status. This information can also be used by the database administrator to override automated recovery procedures by forcing a commit or a rollback to available participating databases.

Usage Considerations

Two phase commit protocols are offered in all modern distributed database products. However, the methods for implementing two phase commits may vary in the degree of automation provided. Some vendors provide a two phase commit implementation that is transparent to the application. Other vendors require specific programming of the calls into an application, and additional programming would be needed should rollback be a requirement; this situation would most likely result in an increase to program cost and schedule.

Maturity

The two phase commit protocol has been used successfully since the 1980s for hotel and airline reservations, stock market transactions, banking applications and credit card systems [Citron 93].

Costs and Limitations

There have been two performance issues with two phase commit:

1. If one database server is unavailable, none of the servers gets the updates. This is correctable if the software administrator forces the commit to the available participants, but if this is a recurring problem the administrator may not be able to keep up, thus causing system and network performance will deteriorate.
2. There is significant demand in network resources as the number of database servers to which data must be distributed increases. This is correctable through network tuning and correctly building the data distribution through database optimization techniques.

Currently, two phase commit procedures are vendor proprietary. There are no standards on how they should be implemented. X/Open has developed a standard that is being implemented in several transaction processing monitors (see pg. 373), but it has not been adopted by the database vendors [X/Open 96]. Two phase commit proprietary protocols have been published by several vendors.

Alternatives

An alternative to updating distributed databases with a two phase commit mechanism is to update multiple servers using a transaction queuing approach where transactions are distributed sequentially. Distributing transactions sequentially raises the problem of users working with different version of the data. In military usage, this could result in planning sorties for targets that have already been eliminated.

Index Categories

Name of technology	Database Two Phase Commit
Application category	Client-Server (AP.2.1.2.1), Data Management (AP.2.6.1)
Quality measures category	Accuracy (QM.2.1.2.1)
Computing reviews category	Distributed Systems (C.2.4)

References and Information Sources

[Citron 93] Citron, A., et al. "Two-Phase Commit Optimization and Tradeoffs in the Commercial Environment," 520-529. *Proceedings of the Ninth International Conference on Data Engineering.* Vienna, Austria, April 19-23, 1993. Los Alamitos, CA: IEEE Computer Society Press, 1993.

[ORACLE7 92] "Two-Phase Commit," 22-1–22-21. *ORACLE7 Server Concept Manual* (6693-70-1292). Redwood City, CA: Oracle, 1992.

[Schussed 96] Schussed, G. *Replication, The Next Generation of Distributed Database Technology* [online]. Available WWW <URL: http://www.dciexpo.com/geos/replica.html> (1996).

[USCB 94] *The Performance of Two-Phase Commit Protocols in the Presence of Site Failures* (TRCS94-09). Santa Barbara, CA: University of California, Computer Science Department, April 1994.

[X/Open 96] X/Open Web Site [online]. Available WWW <URL: http://www.xopen.co.uk/> (1996).

Author

Darleen Sadoski, GTE
sadoski.darleen@mail.ndhm.gtegsc.com

External Reviewer(s)

David Altieri, GTE

Last Modified

10 Jan 97

Defense Information Infrastructure Common Operating Environment

ADVANCED

Note

We recommend Reference Models, Architectures, Implementations—An Overview, pg. 319, as prerequisite reading for this technology description.

Purpose and Origin

The Defense Information Infrastructure (DII) Common Operating Environment (COE) was developed in late 1993. DII COE was designed to eliminate duplication of development (in areas such as mapping, track management, and communication interfaces) and eliminate design incompatibility among Department of Defense (DoD) systems. Conceptually, the COE is designed to reduce program cost and risk through reusing proven solutions and sharing common functionality, rather than developing systems from "scratch" every time. The purpose of DII COE is to field systems with increasing *interoperability*, *reusability*, *portability*, and operational capability, while reducing development time, technical obsolescence, training requirements, and life-cycle cost.

DII COE reuses proven software components contributed by services and programs to provide common Command, Control, Communication, Computer and Intelligence (C4I) functions. For more details on DII COE see the *Defense Information Infrastructure (DII) Common Operating Environment (COE) Integration and Runtime Specification* and the *DII COE Style Guide* [DII COE 96a, DII COE 96b].

Technical Detail

DII COE technically is

- an architecture (including a set of guidelines and standards)
- a runtime environment
- software (including reusable components)
- a definition for acceptable application programming interfaces

The four major areas are described in further detail below:

1. *Architecture.* The DII COE architecture is fully compliant with the Department of Defense's Technical Architecture for Information Management (TAFIM) reference model (see pg. 361). The DII COE architecture, presented in Figure 9, is a "plug and play," client/server architecture (implemented and running) that defines COE interfaces and how system components will fit together and interact.

2. *Runtime environment.* A runtime operating environment that includes a standard user system interface, operating system, and windowing environment. The DII COE architecture facilitates a developer in establishing the environment such that there is no conflict with other developers' products.

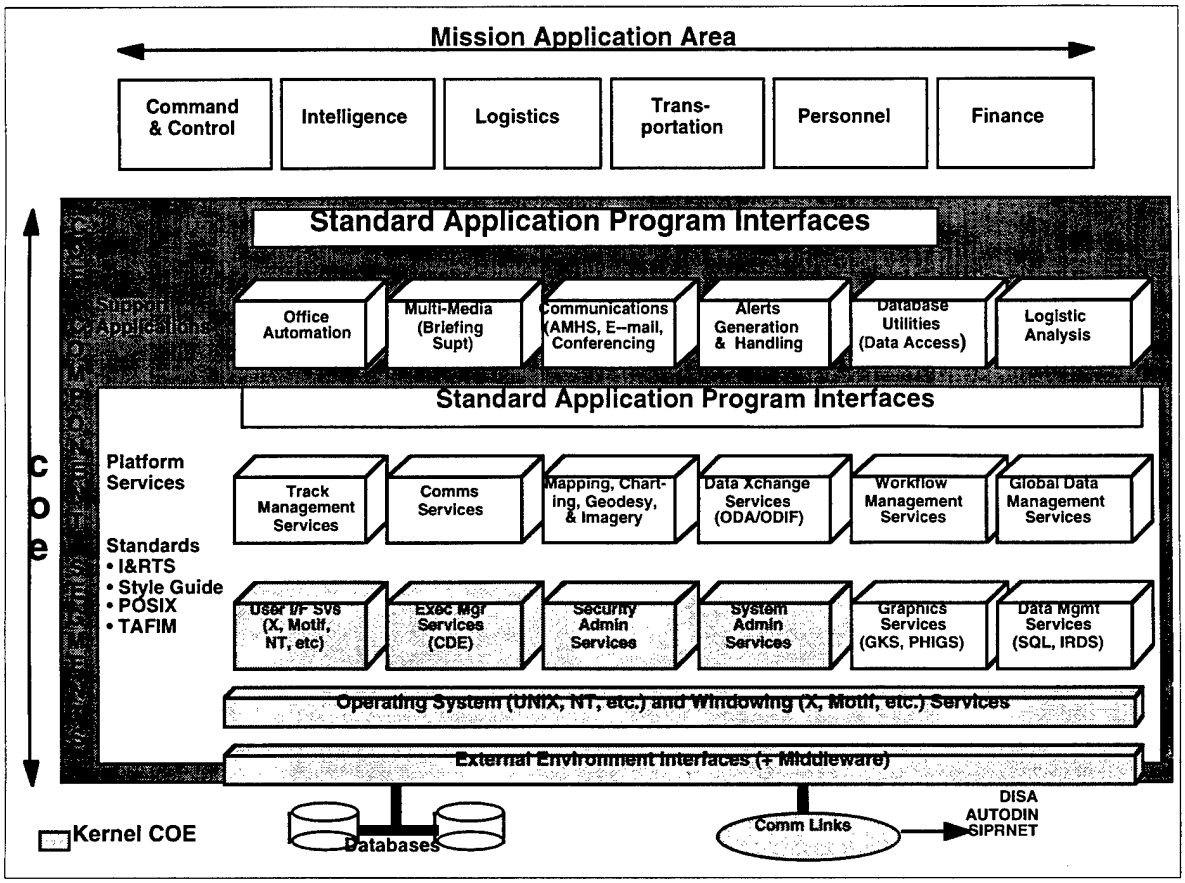


Figure 9: Defense Information Infrastructure Common Operating Environment [DII COE 96]

3. *Software.* A defined set of reusable functions that are already built (available commercially or as government products). Software (with the exception of the operating system and basic windowing software) is packaged in self-contained, manageable units called segments. Segments are the DII COE building block for constructing COE systems. Segments (mission applications and components) may consist of one or more Computer Software Configuration Items (CSCIs). Segments that are part of the reusable (by many mission applications) COE are referred to as COE component segments. Segments are named according to their meaning to operators, rather than internal software structures. Structuring the software into segments allows functionality to be easily added or removed from the target system to meet specific mission and site needs. DII COE databases are divided among segments (as are mission applications) according to the data they contain and the mission applications they support.

The kernel COE (light gray shading in Figure 9) is the minimal set of software that is required on every workstation. It includes operating system, windowing services, and external environment interfaces. There are normally five other services also included in the COE kernel: system administration, security administration, executive manager, and two templates, one for creating privileged operator accounts, and one for creating non-privileged operator accounts. A subset of the kernel COE (defined as Bootstrap COE) is used during initial installation of COE. DII COE is hardware-independent and will run on any open system platform with a standards-based operating system, such as POSIX-compliant UNIX and Windows NT.

4. *APIs.* Two types of application programming interfaces (APIs) (see pg. 79) are defined for accessing COE segments:
   - public APIs (COE interfaces that will be supported for the COE life cycle)
   - private APIs (interfaces that are supported for a short period of time to allow legacy systems to migrate to full COE compliance)

   Newly-developed software (segments) must use public APIs to be COE compliant. The incremental implementation strategy for DII COE is to protect legacy system functionality while migrating to fully-compliant COE design by evolving from private APIs to public APIs.

Usage Considerations

There is only one COE available for use by other systems. This COE is currently being used by GCCS (Global Command and Control System) and GCSS (Global Combat Support System). Any system built to the COE infrastructure must access the services using the COE APIs. This improves interoperability between systems because the integration approach, the tool sets, and the segments (software components, not just algorithms) are used by each system [DII COE 96a].

Conceptually, compliance to COE standards ensures that software that is developed or modified for use within COE meets the intended requirements and goals and will evolve with the COE system. Another perspective is that compliance measures the degree to which "plug and play" is possible [Perry 96]. Owners of legacy systems should be familiar with COE compliance requirements to ensure that scoping and planning for future legacy enhancement includes COE requirements and goals.

There are a number of tradeoffs an organization must address when determining evolution of a legacy system to a system that meets COE compliance.

- What are the goals of the legacy system, and will migrating to COE compliance support achievement of the long range goals?
- What level of COE compliance will best and most cost effectively achieve the legacy system's long range goals?

- What is the current state of the legacy system— how compliant is it today?
- Given the current state of the legacy system, what resources are available to begin and follow through on the migration of the code to COE compliance?
- Does the organization want/need to control the legacy system code, and if not, when in the migration to COE is turning it over to DISA desirable?

Based on this analysis, the appropriate level and strategy for compliance can be determined.

Technically, the four DII COE compliance categories are described below:

Category	Name	Description
1	Runtime Environment	Measures compliance of the proposed segment's fit within the COE executing environment, the amount it reuses COE segments, whether it will run on a COE platform, and whether it will interfere with other segments. This can be done by prototyping within the COE.
2	Style Guide	Measures compliance of the proposed segment's user interface to the Style Guide [DII COE 96b]. This is to ensure that proposed segment will appear consistent with the rest of the COE-based system to minimize training and maintenance cost. Style Guide compliance can be done via a checklist based on the Style Guides requirements.
3	Architectural Compatibility	Measures compliance of the proposed segment's fit within the COE architecture, and the segment's potential life cycle as COE evolves. This can be done by evaluating the segment's use of TAFIM and COE standards and guidelines, and it's internal software structures.
4	Software Quality	Assesses a proposed segment's program risk and software maturity through the use of traditional software metrics. This can be done using measurements such as lines of code and McCabe complexity metrics (see Cyclomatic Complexity, pg. 145).

Category 1 (Runtime) compliance progresses through eight (8) levels of integration from a state of coexistence (agreement on a set of standards and ensure non-interference) with other COE segments, to federated (non-interference when on the same workstation), to fully integrated (share the same software and data). For a segment to be COE compliant, it must be qualified with a category name and compliance level. The following summarizes Category 1's eight levels of compliance; Appendix B of [DII COE 96a] provides a compliance *checklist* for each of the eight

levels. Checklists are the current means of assessing progress toward compliance.

- *Standards Compliance Level One* - A proposed segment shares only a common set of standards with the rest of the COE environment, data sharing is undisciplined, and software reuse is minimal other than use of Commercial-Off-The Shelf (COTS) software products. Level 1 allows simultaneous execution of two systems.
- *Network Compliance Level Two* - Two segments will coexist on the same Local Area Network (LAN), but on different CPUs. There is limited data sharing and there may be common user interface "look and feel" if common user interface standards are applied.
- *Workstation Compliance Level Three* - Two applications can reside on the same LAN, share data, and coexist on the same workstation (environmental conflict have been resolved). The kernel COE, or its functional equivalent, resides on the workstation. Some COE components may be reused, but segmenting may not be done. Segments may not interoperate, and do not use the COE services.
- *Bootstrap Compliance Level Four* - Segment formatting is used in all applications. Segments share the bootstrap COE. Some segment conflicts can be automatically checked by the COE system. COE services are not being used. To switch between segments, users may still require separate login accounts. To submit a prototype to DISA for consideration of use, Bootstrap Compliance is required, although these segments will not be fielded or put in the DISA maintained online library.
- *Minimal COE Compliance Level Five* - All segments share the same kernel COE (equivalent functionality is not acceptable at Level Five). Functionality is available through the COE Executive Manager. Segments may be successfully installed and removed through COE installation tools. Segment descriptor files describe boot, background, and local processes. Segments are registered and available through the online library. Applications appear integrated to the user, but there may be duplication of functionality. Interoperability is not guaranteed. DISA may allow Minimal COE Compliance segments to be installed and used as prototypes at a few sites for evaluation. They can be placed in the library. Currently, Level 5 appears to be the level many legacy systems are targeting.
- *Intermediate COE Compliance Level Six* - Segments use existing account groups, and reuse one or more COE segments. Minor differences may exist between the Style Guide [DII COE 96b] and the segment's graphical user interface implementation.
- *Interoperability Compliance Level Seven* - To ensure interoperability, proposed segments must reuse COE segments, including communication interfaces, message parsers, database tables, track data elements, and logistic services. Public APIs provide access with very few, if any, private APIs. There is no duplicate functionality in the COE segments. DISA requires Interoperability Compliance, for

fieldable products and a migration strategy to full COE Compliance (Level 8). A migration strategy is not needed if the proposed segment will be phased out in the near term.

- *Full COE Compliance Level Eight* - All proposed new segments use COE services to the maximum extent possible. New segments are available through the Executive Manager and are completely integrated into the system. All segments fully comply with the Style Guide. [DII COE 96b]. All segments use only public APIs. There is no duplication of functionality any where in the system (as COE or as a mission application).

Two important resources for COE developers and operational sites are the online COE Software Repository System (CSRS) that is used to disseminate and manage software, and the COE Information Server (CINFO) that is used for documentation, meeting notices and general COE information. [DII COE 96a]

Maturity

COE initial proof of concept was created and installed in 1994 with Global Command and Control System (GCCS) Version 1.0. GCCS version 1.1 was used to monitor events during the 1994 Haiti crisis. In 1995, GCCS version 2.0 began fielding to a number of operational sites.There are two systems currently using DII COE: GCCS (developed in 1994 for a near term replacement for World-Wide Military Command and Control System) and GCSS (already fielded at a number of operational CINCs). It is expected that DII COE will be enhanced to include more functionality in such areas as Electronic Commerce/Electronic Data Interchange (EC/EDI), transportation, base support, personnel, health affairs, and finance. [DII COE 96a]

Costs and Limitations

DII COE is relatively new; actual cost, benefit, and risk information is still being collected.

Dependencies

DII COE is dependent of the evolution of TAFIM to ensure compatibility. (see pg. 361). An additional dependency could be the Joint Technical Architecture (JTA). The JTA is now being mandated as a set of standards and guidelines for C4I systems, specifically in the area of interoperability, to supersede TAFIM Volume 7, which did not appear to go far enough to ensure interoperability [JTA 96].

Alternatives

Under conditions where the TAFIM reference model and DII COE compliance is not required, an alternative model would be the Reference Model for Frameworks of Software Engineering Environments (known as the ECMA reference model [ECMA 93]) that is promoted in Europe, and used commercially and world-wide. Commercially-available Hewlett-Packard products use this model [HP 96]. Another alternative would be

the Common Object Request Broker Architecture (CORBA) if the design called for object-oriented infrastructure (see pg. 107).

Complementary Technologies

Open systems (see pg. 135) would be a complementary technology to DII COE because work done in open system supports the COE goal of achieving interoperable systems.

Index Categories

Name of technology	Defense Information Infrastructure Common Operating Environment
Application category	Software Architecture Models (AP.2.1.1)
Quality measures category	Interoperability (QM.4.1),Reusability (QM.4.4), Portability (QM.4.2)
Computing reviews category	not available

References and Information Sources

[DII COE 96a] *Defense Information Infrastructure (DII) Common Operating Environment (COE) Integration and Runtime Specification (I&RTS)* [online]. Available WWW <URL: http://spider.osfl.disa.mil/dii> (1996).

[DII COE 96b] *DII COE Style Guide*, Version 2.0 [online]. Available WWW <URL: http://spider.osfl.disa.mil/dii> (1996).

[ECMA 93] *Reference Model for Frameworks of Software Engineering Environments*, 3rd Edition (NIST Special Publication 500-211/Technical Report ECMA TR/55). Prepared jointly by NIST and the European Computer Manufacturers Association (ECMA). Washington, DC: U.S. Government Printing Office, 1993.

[HP 96] *Integrated Solutions Catalog for the SoftBench Product Family.* Palo Alto, CA: Hewlett-Packard, 1996.

[JTA 96] U.S. Department of Defense. *Joint Technical Architecture (JTA)* [online]. Available WWW <URL: http://www.itsi.disa.mil/jta.html> (1996).

[Perry 96] Perry, Frank. *Defense Information Infrastructure Common Operating Environment* (briefing). April 17, 1996. Arlington, VA: Defense Information Systems Agency.

Author

Darleen Sadoski, GTE
sadoski.darleen@mail.ndhm.gtegsc.com

Last Modified

10 Jan 97

Distributed/Collaborative Enterprise Architectures ADVANCED

Note

We recommend Client/Server Software Architectures, pg. 101, as prerequisite reading for this technology description.

Purpose and Origin

The distributed/collaborative enterprise architecture emerged in 1993. This software architecture is based on Object Request Broker (ORB) technology (see pg. 291), but goes further than the Common Object Request Broker Architecture (CORBA) (see pg. 107) by using shared, reusable business models (not just objects) on an enterprise-wide scale.[1] The benefit of this architectural approach is that standardized business object models and distributed object computing are combined to give an organization *flexibility*, *scalability*, and *reliability* and improve organizational, operational, and technological effectiveness for the entire enterprise. This approach has proven more cost effective than treating the individual parts of the enterprise. For detailed information on distributed/collaborative enterprise architectures see Shelton and Adler [Shelton 93, Adler 95].

Technical Detail

The distributed/collaborative enterprise architecture allows a business to analyze its internal processes in new ways that are defined by changing business opportunities instead of by preconceived systems design (such as monolithic data processing applications). In this architectural design, an object model represents all aspects of the business; what is known, what the business does, what are the constraints, and what are the interactions and the relationships. A business model is used to integrate and migrate parts of legacy systems to meet the new business profile.

Distributed/collaborative enterprise builds its new business applications on top of distributed business models and distributed computing technology. Applications are built from standard interfaces with "plug and play" components. At the core of this infrastructure is an off-the-shelf, standards-based, distributed object computing, messaging communication component such as an object request broker (ORB) (see pg. 291) that meets Common Object Request Broker Architecture (CORBA) standards (see pg. 107).

1. An enterprise is defined as a system comprised of multiple business systems or multiple subsystems.

This messaging communication hides the following from business applications:

- the implementation details of networking and protocols
- the location and distribution of data, process, and hosts
- production environment services such as transaction management, security, messaging reliability, and persistent storage

The message communication component links the organization and connects it to computing and information resources via the organization's local or wide area network (LAN or WAN). The message communication component forms an enterprise-wide standard mechanism for accessing computing and information resources. This becomes a standard interface to heterogeneous system components.

Usage Considerations

The distributed/collaborative enterprise architecture is being applied in industries and businesses such as banking, investment, trading, credit-granting, insurance, policy management and rating, customer service, transportation and logistics management, telecommunications (long distance, cellular, and operating company), customer support, billing, order handling, product cross-selling, network modeling, manufacturing equipment, and automobiles [Shelton 93].

The most common implementations of objects and object models are written in C++ or Smalltalk. Another popular language for implementing object and object models is Java (see pg. 221).

Available for use in a distributed/collaborative enterprise architecture are products being built to open system standards, operating systems, database management systems, transaction processor monitors, and ORBs. These products are increasingly interchangeable.

Maturity

Since 1993 a number of companies have built and used distributed/collaborative architectures to address their long-term business needs because this model adapts to change and is built according to open system standards [Adler 95].

Costs and Limitations

Distributed/collaborative enterprise architectures are limited by the lack of commercially-available, object-oriented analysis and design method tools that focus on applications (rather than large scale business modeling).

Dependencies

The refinement of CORBA (see pg. 107) and evolution of Object Linking and Embedding (see pg. 271), and the results of standards bodies such

as X/Open [X/Open 96] and Object Management Group (OMG) [OMG 96] will affect the evolution of distributed/collaborative architectures.

Alternatives

Three tier client/server architectures (see pg. 367) are an alternative approach to distributed/collaborative architectures. However, they do not address the need to evolve the business model over time as well as the distributed/collaborative architecture does.

Complementary Technologies

Distributed/collaborative enterprise architectures are enhanced by object-oriented design technologies (see pg. 283).

Index Categories

Name of technology	Distributed/Collaborative Enterprise Architectures
Application category	Client/Server (AP.2.1.2.1)
Quality measures category	Scalability (QM.4.3), Reliability (QM.2.1.2), Maintainability (QM.3.1)
Computing reviews category	Distributed Systems (C.2.4), Software Engineering Design (D.2.10)

References and Information Sources

[Adler 95] Adler, R. M. "Distributed Coordination Models for Client/Sever Computing." *Computer 28,* 4 (April 1995): 14-22.

[Lewis 95] Lewis, T. G. "Where is Client/Server Software Headed?" *Computer 28*, 4 (April 1995): 49-55.

[OMG 96] Object Management Group [online]. Available WWW <URL: http://www.omg.org/tech.htm> (1996).

[Shelton 93] Shelton, Robert E. "The Distributed Enterprise (Shared, Reusable Business Models the Next Step in Distributed Object Computing)." *Distributed Computing Monitor 8*, 10 (October 1993): 1.

[X/Open 96] X/Open [online]. Available WWW <URL: http://www.xopen.co.uk/> (1996).

Author Darleen Sadoski, GTE
sadoski.darleen@mail.ndhm.gtegsc.com

External Reviewer(s) Larry Stafford, GTE

Last Modified 10 Jan 97

Distributed Computing Environment ADVANCED

Note

We recommend Middleware, pg. 251, as prerequisite reading for this technology description.

Purpose and Origin

Developed and maintained by the Open Systems Foundation (OSF), the Distributed Computing Environment (DCE) is an integrated distributed environment which incorporates technology from industry. The DCE is a set of integrated system services that provide an *interoperable* and *flexible* distributed environment with the primary goal of solving interoperability problems in heterogeneous, networked environments.

OSF provides a reference implementation (source code) on which all DCE products are based [OSF 96a].The DCE is *portable* and flexible—the reference implementation is independent of both networks and operating systems and provides an architecture in which new technologies can be included, thus allowing for future enhancements. The intent of the DCE is that the reference implementation will include mature, proven technology that can be used in parts— individual services— or as a complete integrated infrastructure.

The DCE infrastructure supports the construction and integration of client/server applications while attempting to hide the inherent *complexity* of the distributed processing from the user [Schill 93]. The OSF DCE is intended to form a comprehensive software platform on which distributed applications can be built, executed, and maintained.

Technical Detail

The DCE architecture is shown in Figure 10 [Schill 93].

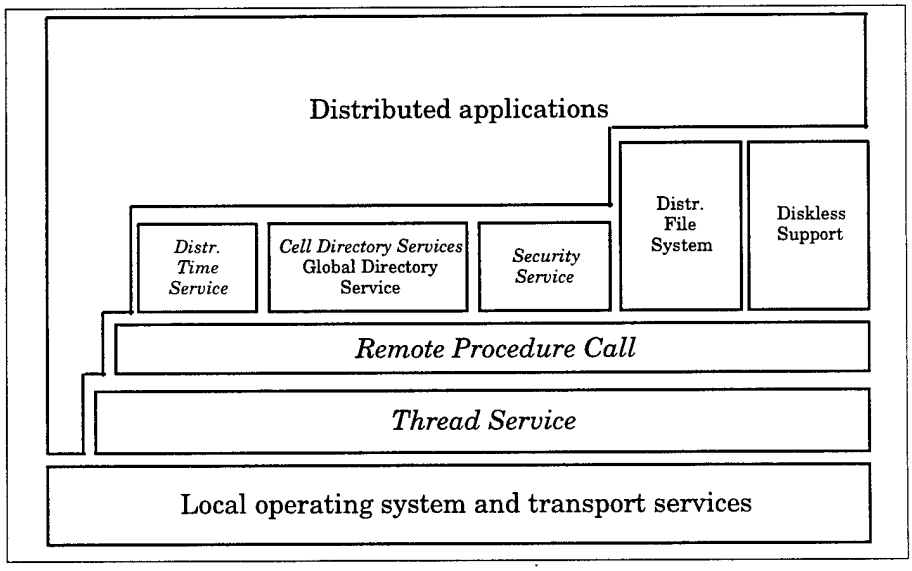


Figure 10: Distributed Computing Environment Architecture

DCE services are organized into two categories:

1. *Fundamental distributed services* provide tools for software developers to create the end-user services needed for distributed computing. They include
   - Remote procedure call (see pg. 323), which provides portability, network *independence*, and *secure* distributed applications.
   - Directory services, which provide full X.500 support and a single naming model to allow programmers and maintainers to identify and access distributed resources more easily.
   - Time service, which provides a mechanism to monitor and track clocks in a distributed environment and accurate time stamps to reduce the load on system administrator.
   - Security service, which provides the network with authentication, authorization, and user account management services to maintain the *integrity*, privacy, and authenticity of the distributed system.
   - Thread service, which provides a simple, portable, programming model for building concurrent applications.
2. *Data-sharing services* provide end users with capabilities built upon the fundamental distributed services. These services require no programming on the part of the end user and facilitate better use of information. They include
   - Distributed file system, which interoperates with the network file system to provide a high-performance, *scalable*, and secure file access system.
   - Diskless support, which allows low-cost workstations to use disks on servers, possibly reducing the need/cost for local disks, and provides performance enhancements to reduce network overhead.

The DCE supports International Open Systems Interconnect (OSI) standards, which are critical to global interconnectivity. It also implements ISO standards such as CCITT X.500, Remote Operations Service Element (ROSE), Association Control Service Element (ACSE), and the ISO session and presentation services. The DCE also supports Internet standards such as the TCP/IP transport and network protocols, as well as the Domain Name System and Network Time Protocol provided by the Internet.

Usage Considerations

The DCE can be used by system vendors, software developers, and end users. It can be used on any network hardware and transport software, including TCP/IP, OSI, and X.25. The DCE is written in standard C and uses standard operating system service interfaces like POSIX and X/Open guidelines. This makes the DCE portable to a wide variety of platforms. DCE allows for the extension of a network to large numbers of

nodes, providing an environment capable of supporting networks of numerous low-end computers (i.e., PCs and Macintosh machines), which is important if downsizing and distributing of processing is desired. Because DCE is provided in source form, it can be tailored for specific applications if desired [OSF 96a].

DCE works internally with the client/server model and is well-suited for development of applications that are structured according to this model. Most DCE services are especially optimized for a structuring of distributed computing systems into a "cell" (a set of nodes/platforms) that is managed together by one authority.

For DCE, intra-cell communication is optimized and relatively secure and transparent. Inter-cell communication, however, requires more specialized processing and more complexity than its intra-cell counterpart, and requires a greater degree of programming expertise.

When using the thread services provided by DCE, the application programmer must be aware of thread synchronization and shared data across threads. While different threads are mutually asynchronous up to a static number defined at initialization, an individual thread is synchronous. The complexity of thread programming should be considered if these services are to be used.

DCE is being used or is planned for use on a wide variety of applications, including the following:

- *The Common Operating Environment.* DCE has been approved by DISA (Defense Information Systems Agency) as the distributed computing technology for the joint services Global Command and Control System (GCCS) Common Operating Environment (COE) (see pg. 155). GCCS will be the single, global command and control, communications, computer, and intelligence system to support the Joint Services.
- *The Advanced Photon Source (APS) system.* This is a synchrotron radiation facility under construction at Argonne National Laboratory.
- *The Alaska Synthetic Aperture Radar Facility (ASF).* This is the ground station for a set of earth-observing radar spacecraft, and is one of the first NASA projects to use DCE in an operational system.
- *The Deep Space Network's Communications Complexes Monitor and Control Subsystem.* This project is deploying DCE for subsystem internal communications, with the expectation that DCE will eventually form the infrastructure of the entire information system.
- *The Multimission Ground Data System Prototype.* This project evaluated the applicability of DCE technology to ground data

systems for support of JPL flight projects (Voyager, Cassini, Mars Global Surveyor, Mars Pathfinder).

- *Earth Observing Systems Data Information System.* This NASA system is one of the largest information systems ever implemented. The system is comprised of legacy systems and data, computers of many varieties, networks, and satellites in space.
- *Command and control prototypes.* MITRE has prototyped command and control (C2) applications using DCE technology. These applications provide critical data such as unit strength, supplies, and equipment, and allow staff officers to view maps of areas of operation [OSF 96b].

Maturity

In early 1992, the OSF released the source code for DCE 1.0. Approximately 12 vendors had ported this version to their systems and had DCE 1.0 products available by June 1993. Many of these original products were "developer's kits" that were not robust, did not contain the entire set of DCE features (all lacked distributed file services), and were suited mostly for UNIX platforms [Chappell 93].

The DCE continues to evolve, but many large organizations have committed to basing their next generation systems on the DCE— over 14 major vendors provided DCE implementations by late 1994, when DCE 1.1 was released.

DCE 1.2.1, released in March 1996, provided the following new features:

- Interface definition language (IDL) support for C++ to include features such as inheritance and object references in support of object-oriented applications. This feature supports adoption of any object model or class hierarchy, thus providing developers with additional flexibility.
- Features to provide for coexistence with other application environments.
- Improvements over DCE 1.1 including enhancements to achieve greater reliability and better performance [OSF 96a].

Two future (post-DCE 1.2) approaches to supporting objects are being considered besides the approach described for DCE 1.2:

1. Installing a CORBA-based product over DCE to provide additional support for distributed object technologies and a wide range of standardized service interfaces.
2. Integrating Network OLE (see pg. 271) into the DCE infrastructure.

Costs and Limitations

DCE was not built to be completely object-oriented. The standard interfaces used by the DCE, as well as all the source code itself, are defined only in the C programming language. For object-oriented applications (i.e., applications being developed using an object-oriented language

(see pg. 287) such as C++ or Ada 95 (see pg. 67)), it may be more complex, less productive (thus more expensive), and less maintainable to use a non-object-oriented set of services like the DCE [Chappell 96].

Object-oriented extensions of the DCE have been developed by industry, but an agreed to vendor-neutral standard was still being worked in 1996.

Dependencies

Dependencies include remote procedure call (RPC) (see pg. 323).

Alternatives

Alternatives include CORBA (see pg. 107), OLE (see pg. 271), and message-oriented middleware (see pg. 247).

Complementary Technologies

DCE, in-part, has been used in building CORBA-compliant (see pg. 107) products as early as 1995. OSF is considering support for objects using Network OLE (see pg. 271).

Index Categories

Name of technology	Distributed Computing Environment
Application category	Distributed Computing (AP.2.1.2)
Quality measures category	Interoperability (QM.4.1), Portability (QM.4.2), Scalability (QM.4.3), Security (QM.2.1.5), Maintainability (QM.3.1), Complexity (QM.3.2.1), Throughput (QM.2.2.3)
Computing reviews category	Distributed Systems (C.2.4)

References and Information Sources

[Brando 96] Brando, T. "Comparing CORBA & DCE." *Object Magazine 6,* 1 (March 1996): 52-7.

[Chappell 93] Chappell, David. "OSF's DCE and DME: Here Today?" *Business Communications Review 23,* 7 (July 1993): 44-8.

[Chappell 96] Chappell, David. *DCE and Objects* [online]. Available WWW <URL: http://www.osf.org/dce/3rd-party/ChapRpt1.html> (1996).

[OSF 96a] Open Software Foundation. *The OSF Distributed Computing Environment* [online]. Available WWW <URL: http://www.osf.org/dce/> (1996).

[OSF 96b] Open Software Foundation. *The OSF Distributed Computing Environment: End-User Profiles* [online]. Available WWW URL: <http://www.osf.org/comm/lit/dce-eup/> (1996).

[Product 96] *OSF DCE Product Survey Report* [online]. Available WWW <URL: http://www.asset.com/WSRD/abstracts/ABSTRACT_1296.html> (1996).

[Schill 93] Schill, Alexander. "DCE—The OSF Distributed Computing Environment Client/Server Model and Beyond," 283. *International DCE Workshop.* Karlsruhe, Germany, October 7-8, 1993. Berlin, Germany: Springer-Verlag, 1993.

Author Cory Vondrak, TRW, Redondo Beach, CA

Last Modified 10 Jan 97

Domain Engineering and Domain Analysis — ADVANCED

Purpose and Origin

The term domain is used to denote or group a set of systems or functional areas, within systems, that exhibit similar functionality. Domain engineering is the foundation for emerging "product line" software development approaches, and affects the *maintainability*, *understandability*, *usability*, and *reusability* characteristics of a system or family of similar systems.

The purpose of this technology description is to introduce the key concepts of domain engineering and provide overview information about domain analysis. Detailed discussions of individual domain analysis methods can be found in the referenced technology descriptions.

Technical Detail

Domain engineering and domain analysis are often used interchangeably and/or inconsistently. Although domain analysis as a term may predate domain engineering, domain engineering is the more inclusive term, and is the process of

- defining the scope (i.e., domain definition)
- analyzing the domain (i.e., domain analysis)
- specifying the structure (i.e., domain architecture development)
- building the components (e.g., requirements, designs, software code, documentation)

for a class of subsystems that will support reuse [Katz 93].

Figure 11 [Foreman 96] shows the process and products of the overall domain engineering activity, and shows the relationships and interfaces of domain engineering to the conventional (individual) system development (application engineering) process. This has come to be known as the two life cycle model.

Domain engineering is related to system engineering, which is an integrated set of engineering disciplines that supports the design, development, and operation of large-scale systems [Eisner 94]. Domain engineering is distinguished from system engineering in that it involves designing assets[1] for a *set or class of multiple applications* as opposed to designing the best solution for a single application. In addition, system engineering provides the "whole solution," whereas domain engineering defines (i.e., limits) the scope of functionality addressed across multiple systems [Simos 96].

1. Examples include requirements, design, history of design decisions, source code, and test information.

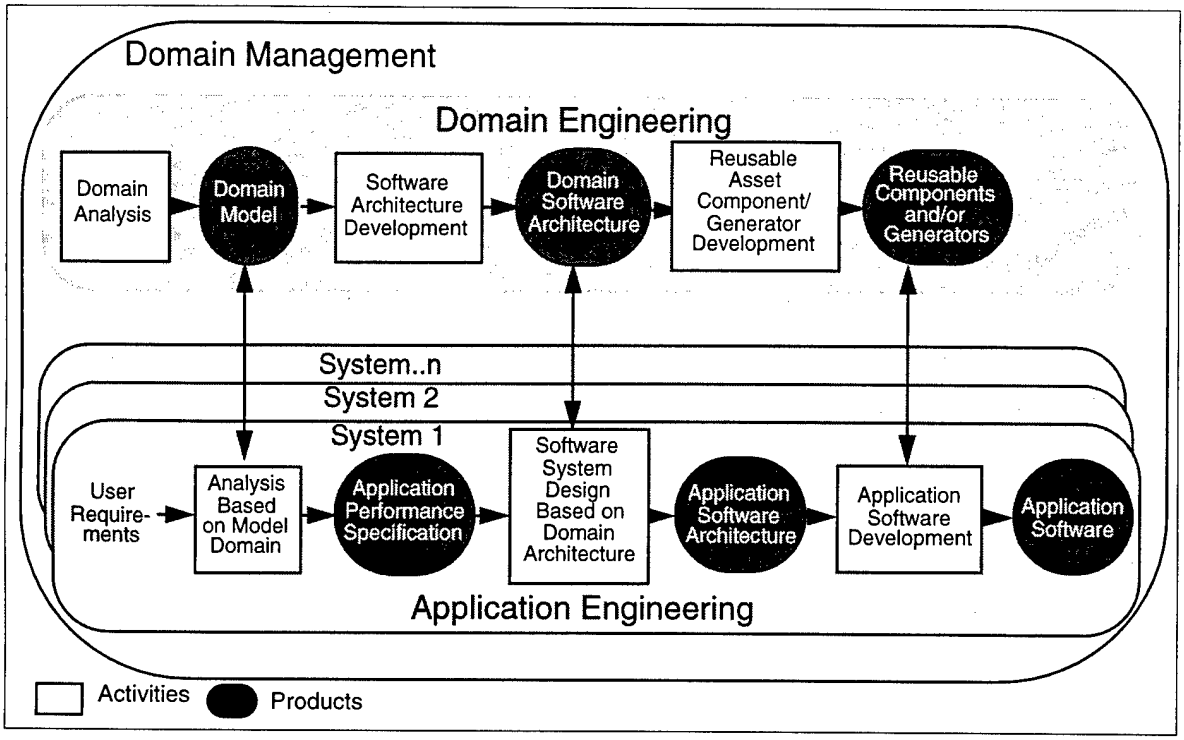


Figure 11: Domain Engineering and Application Engineering (Two Life Cycles)

Domain engineering supports systems engineering for individual systems by enabling coherent solutions across a family of systems: simplifying their construction, and improving the ability to analyze and predict the behavior of "systems of systems" composed of aggregations of those systems [Randall 96].

Domain analysis. Domain analysis (first introduced in the 1980s) is an activity within domain engineering and is the process by which information used in developing systems in a domain is identified, captured, and organized with the purpose of making it reusable when creating new systems [Prieto-Diaz 90]. Domain analysis focuses on supporting systematic and large-scale reuse (as opposed to opportunistic reuse, which suffers from the difficulty of adapting assets to fit new contexts) by capturing both the commonalities and the variabilities[1] of systems within a domain to improve the efficiency of development and maintenance of those systems. The results of the analysis, collectively referred to as a domain model, are captured for reuse in future development of similar systems and in

1. Commonality and variability refer to such items as functionality, data items, performance attributes, capacity, and interface protocols.

maintenance planning of legacy systems (i.e., migration strategy) as shown in Figure 12 [Foreman 96].

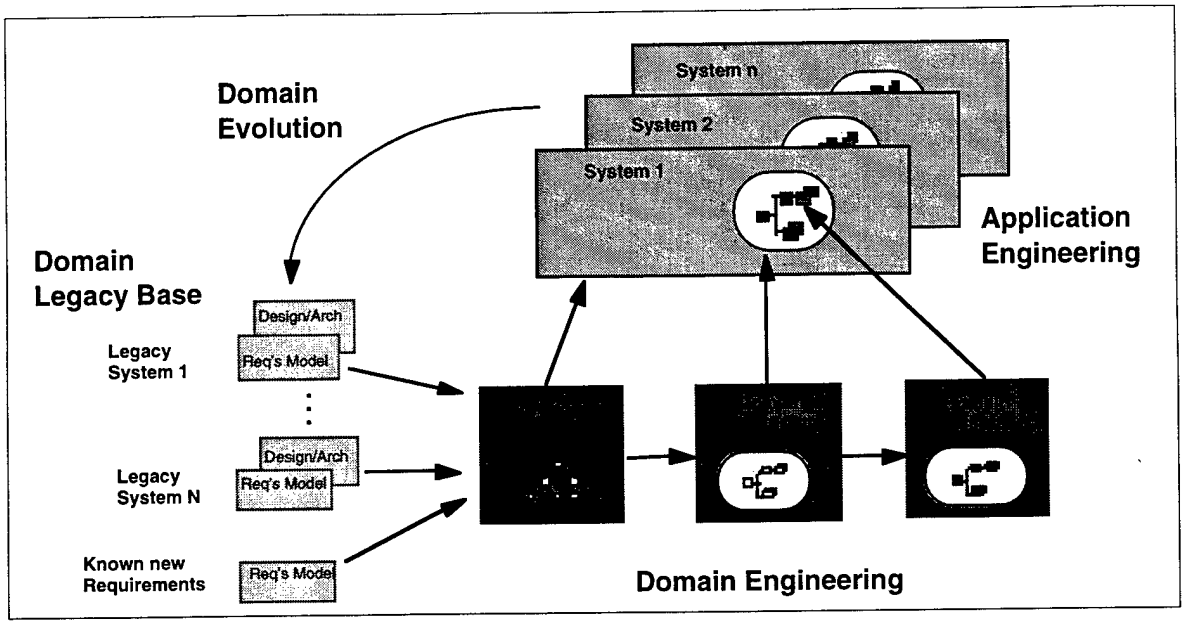


Figure 12: Domain Engineering and Legacy System Evolution

One of the major historical obstacles to reusing a software asset has been the uncertainty surrounding the asset. Questions to be answered included

- How does the software asset behave in its original context?
- How will it behave in a new context?
- How will adaptation affect its behavior [Simos 94]?

Design for reuse techniques (e.g., documentation standards, adaptation techniques) were developed to answer these questions; however, they did not provide the total solution, as a software asset's best scope needed to be determined (i.e., In which set of systems would the software asset be most likely reused?). Domain engineering and analysis methods were developed to answer more global questions, such as:

- Who are the targeted customers for the asset base (the designed collection of assets targeted to a specific domain)?
- Who are the other stakeholders in the domain?
- What is the domain boundary?
- What defines a feature of the domain?
- When is domain modeling complete?

- How do features vary across different usage contexts?
- How can the asset base be constructed to adapt to different usage contexts?

Goals of domain analysis include the following:

- Gather and correlate all the information related to a software asset. This will aid domain engineers in assessing the reusability of the asset. For example, if key aspects of the development documentation (e.g., chain of design decisions used in the development process) are available to a potential reuser, a more cost-effective reuse decision can be made.
- Model commonality and variability across a set of systems. This comparative analysis can reveal hidden contextual information in software assets and lead to insights about underlying rationale that would not have been discovered by studying a single system in isolation. It would answer questions like the following:
  - Why did developers make different design tradeoffs in one system than another?
  - What aspects of the development context influenced these decisions?
  - How can this design history be transformed into more prescriptive guidance to new developers creating systems within this domain?
- Derive common architectures and specialized languages that can leverage the software development process in a specific domain.

There is no standard definition of domain analysis; several domain analysis methods exist. Common themes among the methods include mechanisms to

- define the basic concepts (boundary, scope, and vocabulary) of the domain that can be used to generate a domain architecture
- describe the data (e.g., variables, constants) that support the functions and state of the system or family of systems
- identify relationships and constraints among the concepts, data, and functions within the domain
- identify, evaluate, and select assets for (re-)use
- develop adaptable architectures

Wartik provides criteria for comparing domain analysis methods [Wartik 92]. Major differences between the methods fall into three categories:

- *Primary product of the analysis.* In the methods, the results of the analysis and modeling activities may be represented differently. Examples include: different types of reuse library infrastructures

(e.g., structured frameworks for cataloging the analysis results), application engineering processes, etc.

- *Focus of the analysis.* The methods differ in the extent they provide support for
  - context analysis: the process by which the scope of the domain is defined and analyzed to identify variability
  - stakeholder analysis: the process of modeling the set of stakeholders of the domain, which is the initial step in domain planning
  - rationale capture: the process for identifying and recording the reasoning behind the design of an artifact
  - scenario definition: mechanisms to capture the dynamic aspects of the system
  - derivation histories: mechanisms for replaying the history of design decisions
  - variability modeling: the process for identifying the ways in which two concepts or entities differ
  - legacy analysis: the process for studying and analyzing an existing set of systems
  - prescriptive modeling: the process by which binding decisions and commitments about the scope, architecture, and implementation of the asset base are made
- *Representation techniques.* An objective of every domain analysis method is to represent knowledge in a way that is easily understood and machine-processable. Methods differ in the type of representation techniques they use and in the ease with which new representation techniques can be incorporated within the method.

Examples of domain analysis methods include

- Feature-oriented domain analysis (FODA) (see pg. 185), a domain analysis method based upon identifying the features of a class of systems, defines three basic activities: context analysis, domain modeling, and architecture modeling [Kang 90].
- Organization domain modeling (ODM) (see pg. 297), a domain engineering method that integrates organizational and strategic aspects of domain planning, domain modeling, architecture engineering and asset base engineering [Simos 96].

Randall, Arango, Prieto-Diaz, and the Software Productivity Consortium offer other domain engineering and analysis methods [Randall 96, Arango 94, Prieto-Diaz 91, SPC 93].

Usage Considerations

Domain analysis is best suited for domains that are mature and stable, and where context and rationale for legacy systems can be rediscovered through analysis of legacy artifacts and through consultation with domain experts. In general, when applying a domain analysis method, it is impor-

tant to achieve independence from architectural and design decisions of legacy systems. Lessons learned from the design and implementation of the legacy system are essential; however, the over-reliance on precedented features and legacy implementations may bias new developments.

Maturity

See individual technologies.

Costs and Limitations

See individual technologies.

Complementary Technologies

Use of visual programming techniques can provide better understanding of key software assets like execution patterns, specification and design animations, testing plans, and systems simulation. Other complementary technologies include comparative/taxonomic modeling and techniques for the development of adaptable architectures/implementations (e.g., generation, decision-based composition).

Index Categories

Name of technology	Domain Engineering and Domain Analysis
Application category	Domain Engineering (AP.1.2.4)
Quality measures category	Reusability (QM.4.4), Maintainability (QM.3.1), Understandability (QM.3.2)
Computing reviews category	Software Engineering Tools and Techniques (D.2.2)

References and Information Sources

[Arango 94] Arango, G. "Domain Analysis Methods," 17-49. *Software Reusability.* Chichester, England: Ellis Horwood, 1994.

[Eisner 94] Eisner, H. "Systems Engineering Sciences," 1312-1322. *Encyclopedia of Software Engineering.* New York, NY: John Wiley and Sons, 1994.

[Foreman 96] Foreman, John. *Product Line Based Software Development— Significant Results, Future Challenges.* Software Technology Conference, Salt Lake City, UT, April 23, 1996.

[Hayes 94] Hayes-Roth, F. *Architecture-Based Acquisition and Development of Software: Guidelines and Recommendations from the ARPA Domain-Specific Software Architecture (DSSA) Program.* Palo Alto, CA: Teknowledge Federal Systems, 1994.

[Kang 90] Kang, K., et al. *Feature-Oriented Domain Analysis (FODA) Feasibility Study* (CMU/SEI-90-TR-21, ADA 235785). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1990.

[Katz 94] Katz, S., et al. *Glossary of Software Reuse Terms*. Gaithersburg, MD: National Institute of Standards and Technology, 1994.

[Prieto-Diaz 90] Prieto-Diaz, R. "Domain Analysis: An Introduction." *Software Engineering Notes 15,* 2 (April 1990): 47-54.

[Prieto-Diaz 91] Prieto-Diaz, R. *Domain Analysis and Software Systems Modeling.* Los Alamitos, CA: IEEE Computer Society Press, 1991.

[Randall 96] Randall, Rick. *Space and Warning C^2Product Line Domain Engineering Guidebook*, Version 1.0 [online]. Available WWW <URL: http://source.asset.com/stars/loral/Test/deguidebook/home.htm> (1996). Note: This is a beta version of the document; this URL is subject to change.

[Simos 96] Simos, M., et al. *Software Technology for Adaptable Reliable Systems (STARS) Organization Domain Modeling (ODM) Guidebook* Version 2.0 (STARS-VC-A025/001/00). Manassas, VA: Lockheed Martin Tactical Defense Systems, 1996. Also available [online] WWW <URL: http://www.asset.com/WSRD/abstracts/ABSTRACT_1176.html> (1996).

[SPC 93] *Reuse-Driven Software Processes Guidebook* Version 2.00.03 (SPC-92019-CMC). Herndon, VA: Software Productivity Consortium, 1993.

[Svoboda 96] Svoboda, Frank. *The Three "R's" of Mature System Development: Reuse, Reengineering, and Architecture* [online]. Available WWW <URL: http://source.asset.com/stars/darpa/Papers/ArchPapers.html> (1996).

[Wartik 92] Wartik, S. & Prieto-Diaz, R. "Criteria for Comparing Reuse-Oriented Domain Analysis Approaches." *International Journal of Software Engineering and Knowledge Engineering 2,* 3 (September 1992): 403-431.

Author Liz Kean, Rome Laboratory
liz@se.rl.af.mil

External Reviewer(s) Jim Baldo, MITRE, Washington, DC
Dick Creps, Lockheed Martin, Manassas, VA
Teri Payton, Lockheed Martin, Manassas, VA
Spencer Peterson, SEI
Rick Randall, Kaman Sciences, Colorado Springs, CO
Mark Simos, Organon Motives, Belmont, MA

Last Modified 10 Jan 97

Feature-Based Design Rationale Capture Method for Requirements Tracing

ADVANCED

Purpose and Origin

A design rationale is a representation of the reasoning behind the design of an artifact. The purpose of a feature-based design rationale capturing method is to provide mechanisms to track for each feature of the system; the description of the engineering decision that the feature represents includes

- the summary of the tradeoffs that were considered in arriving at the decision
- the ultimate rationale for the decision

The idea for feature-based design rationale capture originated during the performance of domain analysis in a software development project [Bailin 90]. The need to reverse engineer the rationales for various decisions suggested that a reuse environment should not simply present to the developer a set of alternative architectures that have been used on previous systems. It is necessary to present the rationales and issues involved in choosing among the alternatives. The feature-based approach evolved from the argumentation-based design rationale capture methods (see pg. 91). The major difference between the approaches is that the knowledge is organized around distinctive features of a system (feature-based) rather than around issues raised during the development process (argument-based).

Replaying the history of design decisions facilitates the understanding of the evolution of the system, identifies decision points in the design phase where alternative decisions could lead to different solutions, and identifies dead-end solution paths. The captured knowledge should enhance the *evolvability* of the system and the *reusability* of components in the system.

Technical Detail

In the feature-based design rationale capture method, a feature is any distinctive or unusual aspect of a system, or a manifestation of a key engineering decision [Bailin 90]. (Note: The definition of a feature in this context is different from a feature in feature-oriented domain analysis (FODA) (see pg. 185), in which a feature is a user-visible aspect or characteristic of the domain [Kang 90].) The features in a system make this system different from any other system in the domain. Examples of categories of features are: operational, interface, functional, performance, development methodology, design, and implementation. Each feature has a list of tradeoffs and rationale associated with it. Representations of the set of features may be entity relationship, dataflow, object communi-

cation, assembly, classification, stimulus-response, and state transition diagrams. The purpose of the multiple representations or views is to add flexibility in responding to evolving design paradigms, life cycle models, etc. A new way of looking at a system can be represented by adding a new view or way of looking at the features of the system. This provides a uniqueness and strength to this method that does not exist in other design rationale capturing methods. This approach makes the software engineering process become a process of answering questions about the features of a system rather than a cookbook-like procedure defined by a particular development method.

Usage Considerations

The use of this technology is oriented toward the entire organization, rather than single projects, because the big payoff occurs when a substantial database of corporate knowledge is organized and maintained. If an organization builds the same types of systems, the knowledge acquired in previous developments can be reused. Since the organization of information is around the features of a system as opposed to the issues that arise during a development project, only the issues that observably affect the content of the resulting system are saved.

The use of this technology requires the development of a shared, consistent, and coherent policy by a project team. A procedure for overall coordination must be developed.

Maturity

To date, there is at least one commercially-available tool to support the feature-based design rationale capture method. It is not a highly automated tool, but rather a bookkeeper to support an experience-based, learning-based development process. The commercial tool is based upon a prototype that has been used in laboratory experiments. The feature-based design rational capture method was used on the Software Technology for Adaptable, Reliable Systems (STARS) program to support the Organization Domain Modeling (ODM) process (see pg. 297) [Lettes 96].

Costs and Limitations

Feature-based design rationale capture methods and supporting tools require additional time and effort throughout the software life cycle. The system is described using multiple views that must be generated and maintained throughout the life of the project. Depending upon the size of the system, the number of views could be large. There is no integrated view of the system and this must be accomplished either mentally by the engineers on the project or through the use of an additional tool/technique. Training for the project team as well as the potential reuser is essential to make effective use of the method.

Alternatives

There are several alternative approaches to requirements traceability methods. Examples include Argument-Based Design Rationale Capture Methods for Requirements Tracing (see pg. 91), an approach centered around the debate process (i.e., arguments and their resolution) that occurs during requirements analysis, and the Process Knowledge Method, an extension of the argument-based approach that includes a formal representation to provide two way traceability between requirements and artifacts and facilities for temporal reasoning (i.e., mechanisms to use the captured knowledge).

Index Categories

Name of technology	Feature-Based Design Rationale Capture Method for Requirements Tracing
Application category	Requirements Tracing (AP.1.2.3), Domain Engineering (AP.1.2.4)
Quality measures category	Completeness (QM.1.3.1), Consistency (QM.1.3.2), Traceability (QM.1.3.3), Effectiveness (QM.1.1), Reusability (QM.4.4), Understandability (QM.3.2), Maintainability (QM.3.1)
Computing reviews category	Software Engineering Tools and Techniques (D.2.2), Software Engineering Design (D.2.10)

References and Information Sources

[Bailin 90] Bailin, S., et al. "KAPTUR: Knowledge Acquisition for Preservation of Tradeoffs and Underlying Rationale," 95-104. *Proceedings of the 5th Annual Knowledge-Based Software Assistant Conference*. Liverpool, NY, September 24-28, 1990. Rome, NY: Rome Air Development Center, 1990.

[Gotel 95] Gotel, Orlena. *Contribution Structures for Requirements Traceability*. London, England: Department of Computing, Imperial College, 1995.

[Kang 90] Kang, Kyo C., et al. *Feature-Oriented Domain Analysis (FODA) Feasibility Study* (CMU/SEI-90-TR-21, ADA235785). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1990.

[Lettes 96] Lettes, Judith A. & Wilson, John. *Army STARS Demonstration Project Experience Report* (STARS-VC-A011/003/02). Manassas, VA: Loral Defense Systems-East, 1996.

[Ramesh 92] Ramesh, Balasubramaniam & Dhar, Vasant. "Supporting Systems Development by Capturing Deliberations During Requirements Engineering." *IEEE Transactions on Software Engineering 18,* 6 (June 1992): 498-510.

[Shum 94] Shum, Buckingham Simon & Hammond, Nick. "Argumentation-Based Design Rationale: What Use at What Cost?" *International Journal of Human-Computer Studies 40,* 4 (April 1994): 603-52.

Author Liz Kean, Rome Laboratory
liz@se.rl.af.mil

Last Modified 10 Jan 97

Feature-Oriented Domain Analysis

ADVANCED

Note

Domain Engineering and Domain Analysis, pg. 173, provides overview information about domain analysis.

Purpose and Origin

Feature-oriented domain analysis (FODA) is a domain analysis method based upon identifying the prominent or distinctive features of a class of systems. FODA resulted from an in-depth study of other domain analysis approaches [Kang 90]. FODA affects the *maintainability*, *understandability*, and *reusability* characteristics of a system or family of systems.

Technical Detail

The FODA methodology was founded on two modeling concepts: abstraction and refinement. Abstraction is used to create domain products from the specific applications in the domain. These generic domain products abstract the functionality and designs of the applications in a domain. The generic nature of the domain products is created by abstracting away "factors" that make one application different from other related applications. The FODA method advocates that applications in the domain should be abstracted to the level where no differences exist between the applications. Specific applications in the domain are developed as refinements of the domain products.

Domain Engineering and Domain Analysis, pg. 173, identifies three areas to differentiate between domain analysis methods. Distinguishing features for FODA include the following:

Primary Product of the Analysis. The primary product of FODA is a structured framework of related models that catalog the domain analysis results.

Focus of Analysis. The FODA process is divided into three phases:

- *Context analysis.* The purpose of context analysis is to define the scope of a domain. Relationships between the domain and external elements (e.g., different operating environments, different data requirements, etc.) are analyzed, and the variabilities are evaluated. The results are documented in a context model (e.g., block diagram, structure diagram, dataflow diagram, etc.).
- *Domain modeling.* Once the domain is scoped, the domain modeling phase provides steps to analyze the commonalities and differences addressed by the applications in the domain and produces several domain models. The domain modeling phase consists of three major activities:
  - Feature analysis. During feature analysis, a customer's or end user's understanding of the general capabilities or features of the class of systems is captured. The features,

which describe the context of domain applications, the needed operations and their attributes, and representation variations are important results because the features model generalizes and parameterizes the other models produced in FODA. Examples of features include: function descriptions, descriptions of the mission and usage patterns, performance requirements, accuracy, time synchronization, etc. Features may be defined as alternative, optional, or mandatory. Mandatory features represent baseline features and their relationships. The alternative and optional features represent the specialization of more general features (i.e., they represent what changes are likely to occur in different circumstances). For optimal benefit, the resulting features model should be captured in a tool with access to rule-based language(s) so dependencies among features can be maintained and understood.

- Information analysis. During information analysis, the domain knowledge and data requirements for implementing applications in the domain are defined and analyzed. Domain knowledge includes relevant scientific theory and engineering practice, capabilities and uses of existing systems, past system development and maintenance experience and work products, design rationales, history of design changes, etc. The purpose of information analysis is to represent the domain knowledge in terms of domain entities and their relationships, and to make them available for the derivation of objects and data definitions during operational analysis and architecture modeling. The information model may be of the form of an entity relationship (ER) model, a semantic network, or an object-oriented (OO) model.
- Operational analysis. During operational analysis, the behavioral characteristics (e.g., dataflow and control-flow commonalities and differences, finite state machine model) of the applications in a domain are identified. This activity abstracts and then structures the common functions found in the domain and the sequencing of those actions into an operational model. Common features and information model entities form the basis for the abstract functional model. Unique features and information model entities complete the functional model. The control and data flow of an individual application can be instantiated or derived from the operational model with appropriate adaptation.

- *Architecture Modeling.* This phase provides a software solution for applications in the domain. An architectural model, which is a high-level design for applications in a domain, is developed. It focuses on identifying concurrent processes and domain-oriented common modules. It defines the process for allocating the features, functions,

and data objects defined in the domain models to the processes and modules.

Representation Techniques. The use of COTS methods or tools must be integrated on a case-by-case basis. Currently FODA has been integrated with tools that support object-oriented models, entity relationship models, and semantic networks.

Usage Considerations

Based upon early pilot projects applying the FODA method [Kang 90, Cohen 92], the following lessons learned should be considered:

- A clear definition of the users of the domain model is essential. They should be well-defined during the context analysis phase.
- Early identification of the domain experts and sources of information is important. Effectively working with domain experts is the best means to achieving adoption of the domain model by potential users.
- The need for automated support for the domain modeling phase was identified. No modeling tools that support the FODA approach to ER modeling (i.e., ER + semantic data modeling) exist. Integration with existing modeling capabilities is achieved on a case-by-case basis. FODA was integrated with Hamilton Technologies 001 tool suite [Krut 93]. The integration was not automatic and there were areas where the 001 capabilities did not meet the FODA requirements. These were resolved through workarounds and negotiations with Hamilton Technologies.

Maturity

The FODA method is well-defined and has been applied on both commercial and military applications. It was applied to the

- Army Movement Control Domain [Cohen 92]
- In-Transit Visibility Modernization (ITVMOD) domain analysis effort [Petro 95, Devasirvatham 94]
- Telecommunication Automated Prompt and Response Domain at NORTEL (Northern Telecom) [Schnell 96]

Training is available.

Costs and Limitations

For small projects, use of the simulation capabilities of a commercial tool like Statemate was effective during operational analysis in demonstrating the capabilities of a system; however, for large projects potential users must be convinced that the model and tool can be effectively used to specify a new system of the scale needed. The ability to use a modeling tool that can both capture the domain model and produce prototype code to simulate a system based upon feature selection would benefit the FODA method.

Index Categories

Name of technology	Feature-Oriented Domain Analysis
Application category	Domain Engineering (AP.1.2.4)
Quality measures category	Reusability (QM.4.4), Maintainability (QM.3.1), Understandability (QM.3.2)
Computing reviews category	Software Engineering Tools and Techniques (D.2.2)

References and Information Sources

[Cohen 92] Cohen, Sholom G., et al. *Application of Feature-Oriented Domain Analysis to the Army Movement Control Domain* (CMU/SEI-91-TR-28, ADA 256590). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1992.

[Devasirvatham 94] Devasirvatham, Josiah, et al. *In-Transit Visibility Modernization Domain Scoping Report Comprehensive Approach to Reusable Defense Software* (STARS-VC-H0002/001/00). Fairmont, WV: Comprehensive Approach to Reusable Defense Software, 1994.

[Kang 90] Kang, Kyo C., et al. *Feature-Oriented Domain Analysis (FODA) Feasibility Study* (CMU/SEI-90-TR-21, ADA 235785). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1990.

[Krut 93] Krut, Robert W. Jr. *Integrating 001 Tool Support into the Feature-Oriented Domain Analysis Methodology* (CMU/SEI-93-TR-11, ESC-TR-93-188) Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1993.

[Krut 96] Krut, R. & Zalman, N. *Domain Analysis Workshop Report for the Automated Prompt & Response System Domain* (CMU/SEI-96-SR-001). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1996.

[Peterson 91] Peterson, A. Spencer & Cohen, Sholom G. *A Context Analysis of Movement Control Domain for the Army Tactical Command and Control System* (CMU/SEI-91-SR-03). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1991.

[Petro 95] Petro, James J.; Peterson, Alfred S.; & Ruby, William F. *In-Transit Visibility Modernization Domain Modeling Report Comprehensive Approach to Reusable Defense Software* (STARS-VC-H002a/001/00). Fairmont, WV: Comprehensive Approach to Reusable Defense Software, 1995.

[Schnell 96] Schnell, K.; Zalman, N.; & Bhatt, Atul. *Transitioning Domain Analysis: An Industry Experience* (CMU/SEI-96-TR-009). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1996.

Author Liz Kean, Rome Laboratory
liz@se.rl.af.mil

External Reviewer(s) Spencer Peterson, SEI

Last Modified 10 Jan 97

Firewalls and Proxies

IN REVIEW

Note

We recommend Computer System Security— an Overview, pg. 129, as prerequisite reading for this technology description.

Purpose and Origin

Firewalls and proxies were developed in the early 1990s as the use of the Internet rapidly expanded. Malicious users on the Internet often try to break into computers on a network to obtain information illegally or to cause damage. There can also be malicious users on Intranets and LANs. In addition to the security provided within each host on the network to prevent malicious access, firewalls and proxies are a means of providing security between networks (see Figure 22, pg. 310). A firewall is one of several ways of protecting one network from another network [Vacca 96]. Proxies are application programs that masquerade as other programs to provide security checks. They frequently masquerade as email, FTP, Telnet, or World Wide Web (WWW) clients.

Technical Detail

The fundamental function of a firewall is to restrict the flow of information between two networks [Garfinkel 96]. As shown in Figure 13, there are three basic firewall configurations:

1. screening router
2. screened host
3. sub network

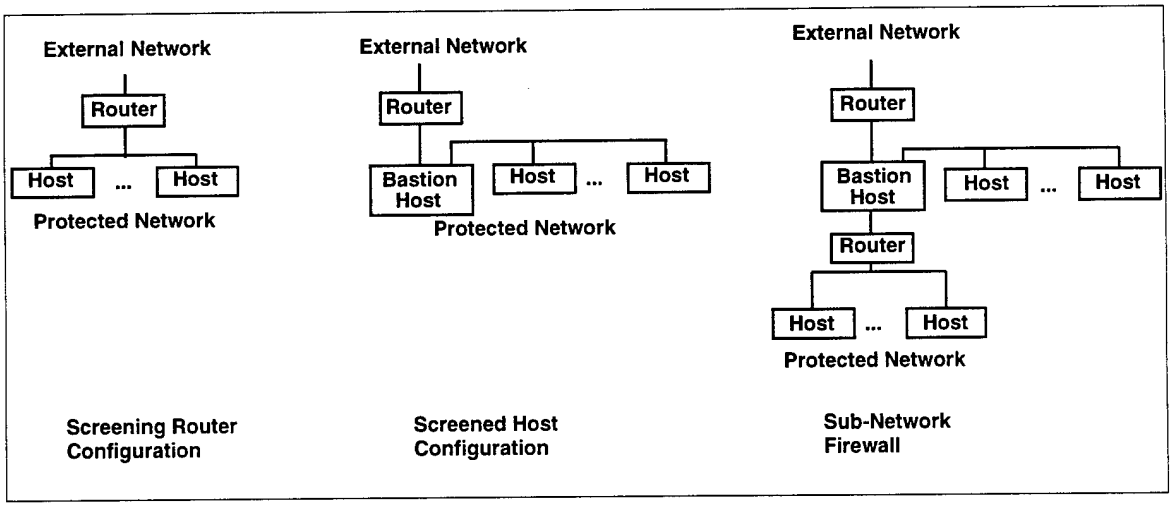


Figure 13: Firewall Configurations

The simplest scheme is the Screening Router. Networks are connected to other networks through hardware called a router. Sophisticated routers can be set up to operate as a packet filter. They only allow Internet Pro-

tocol (IP) packets that have a predetermined destination or source ID to pass to the protected network. This scheme is sometimes called the Packet Filtering firewall.

The Screened Host is a more secure scheme. It consists of a screening router that only routes to a single host on the protected network, which then determines whether the data can be passed on to other hosts in the network. That host is sometimes called the Bastion Host. This host examines incoming traffic to determine if it can be passed to one of the other hosts on the protected network, and examines outgoing traffic to determine if it can be permitted to leave. In addition to the filtering software that executes on the Bastion Host, proxies may execute. This software allows for programs like Telnet and FTP, that have their own applications protocol, to execute that protocol across the firewall and still be secure. For example, a proxy for FTP can examine an incoming FTP block and, if approved, pass it on to the FTP code that executes in one of the protected hosts. A variation of the Screened Host scheme is to eliminate the router. The Bastion Host is connected directly to the external network and the protected network and performs the screening function of the missing router in software. This variation of the Screened Host configuration is called the Dual-Homed Gateway.

The most secure firewall scheme is the Sub-Network Firewall scheme. It consists of a network of hosts (there may be only one host) that is isolated from the external network with a router and also isolated from the protected network with another router. This sub-network is sometimes called the Perimeter Network. The hosts in the sub-network may include applications like FTP or Telnet that are protected by proxies. Individual hosts may be used to service separate protocols— e.g., one host services FTP transactions, another services Telnet transactions. The router between the external network and the sub-network performs the same function as the router in the Screened Host scheme. It routes traffic that is allowed to pass to a Bastion Host in the sub-network. The router between the sub-network and the protected network provides protection if the Bastion Host is penetrated. It will not allow packets that did not originate from the Bastion Host in the sub-network through to the protected network.

Usage Considerations

The least expensive and fastest form of a firewall is the Screening Router. The Screening Router allows the most free Internet access from within the protected network. This scheme is most likely to be used where minimum security is required, for example, on an administrative network, as it is most susceptible to penetration. The Screened Host Gateway firewall scheme is more secure but is also more expensive than the Screening Router, as it ties up one of the hosts on the protected network. The

Sub-Network is the most secure firewall scheme but also requires the most hardware and is the most difficult to administer.

Maturity

Many commercial products are available to implement firewalls. Routers and Gateway software capabilities have come into widespread use with the explosive growth of the Internet that began in the early 1990s. The capabilities provided by these products must be constantly reevaluated by network managers since the threat to network security is constantly getting more sophisticated. A large number of Internet hackers are constantly seeking ways to circumvent current technology. The most mature firewall scheme is the Dual-Homed Gateway, which used a commercially-available host connected to two networks before sophisticated routers were developed.

Costs and Limitations

The Screening Router firewall scheme is inexpensive but also has the most security limitations. Routers used in this scheme have no software, so logging of activity through the router (that may be break-in attempts) is not feasible. Individualized filtering by users is not possible. Since detection of break-ins is difficult, the hosts on the protected network may be thoroughly compromised before a successful break-in attempt is known. Since any of the hosts in the protected network can be accessed through the router, the network is only as secure as the weakest host on the network. Once that host is broken into it can be used to access the other hosts.

Since the Screened Host firewall by design disables the UNIX capability to automatically forward received packets, it complicates the processing of application level protocols for programs like FTP and Telnet. In some cases, Screened Host firewalls prevent those services from being used. The Bastion Host requires particular effort to make it the most secure host on the protected network because if it is broken into the other hosts are all vulnerable.

The Sub-Network scheme requires the most hardware and is the most difficult to administer because the addresses on two routers must be maintained. The software for the Bastion host in the network is more complex since it must have proxies to provide many Internet services to the protected network.

Dependencies

Firewall technology is driven by the capabilities of the rapidly-changing networking technologies. For instance, when Java (see pg. 221) applets became available on the World Wide Web (WWW), it was possible to import malicious code hidden in the applets. To prevent this, it was desirable to block any Java applet at the firewall. If a proxy was being used in

the firewall to filter WWW traffic, the proxy had to be enhanced to recognize Java applets from the WWW protocol.

Alternatives

The security alternative to using Firewalls to prevent theft of data or damage from malicious network users is physical isolation of the networks. That may conflict with mission performance needs in a C4I environment if manual transfer of data from network to network is not acceptable. Data theft may be prevented through the encryption of data, but that will not stop malicious damage.

Complementary Technologies

Network security guards are a complimentary technology as they perform similar functionality in a trusted environment. Intrusion detection technology (see pg. 217) may be incorporated into Firewall software.

Index Categories

Name of technology	Firewalls and Proxies
Application category	System Security (AP.2.4.3)
Quality measures category	Vulnerability (QM.2.1.4.1), Security(QM.2.1.5)
Computing reviews category	Security & Protection (K.6.5), Computer-Communications Network Security and Protection (C.2.0)

References and Information Sources

[Garfinkel 96] Garfinkel, Simson & Spafford, Gene. *Practical UNIX and Internet Security* Second Edition. Sebastopol, CA: O'Reilly & Associates, Inc., 1996.

[Vacca 96] Vacca, John. *Internet Security Secrets.* Foster City, CA: IDG Books Worldwide, Inc., 1996.

Author

Tom Mills, Lockheed Martin
TMILLS@ccs.lmco.com

Last Modified

10 Jan 97

Function Point Analysis

ADVANCED

Purpose and Origin

The function point metric was devised in 1977 by A. J. Albrecht, then of IBM, as a means of measuring software size and *productivity*. It uses functional, logical entities such as inputs, outputs, and inquiries that tend to relate more closely to the functions performed by the software as compared to other measures, such as lines of code. Marciniak provides a good capsule introduction to the application of function point measurement [Marciniak 94].

Function point definition and measurement have evolved substantially; the International Function Point User Group (IFPUG), formed in 1986, actively exchanges information on function point analysis (FPA) [IFPUG 96]. The original metric has been augmented and refined to cover more than the original emphasis on business-related data processing. FPA has become generally accepted as an effective way to

- estimate a software project's size (and in part, duration)
- establish productivity rates in function points per hour
- evaluate support requirements
- estimate system change costs
- normalize the comparison of software modules

However, uniformity of application and results are still issues (see Usage Considerations on pg. 196). For reasons explained below in Technical Detail, FPA has been renamed functional size measurement, but FPA remains the more commonly used term.

Technical Detail

Basic function points are categorized into five groups: outputs, inquiries, inputs, files, and Interfaces. A function point is defined as one end-user business function, such as a query for an input. This distinction is important because it tends to make a function point map easily into user-oriented requirements, but it also tends to hide internal functions, which also require resources to implement. To make up for this (and other) weaknesses, some refinements to and/or variations of the basic Albrecht definition have been devised, including

- *Early and easy function points.* Adjusts for problem and data complexity with two questions that yield a somewhat subjective complexity measurement; simplifies measurement by eliminating the need to count data elements.
- *Engineering function points.* Elements (variable names) and operators (e.g., arithmetic, equality/inequality, Boolean) are counted. This variation highlights computational function [Umholtz 94]. The

intent is similar to that of the operator/operand-based Halstead measures (see pg. 209).

- *Bang measure.* Defines a function metric based on twelve primitive (simple) counts that affect or show Bang, defined as "the measure of true function to be delivered as perceived by the user" [DeMarco 82]. Bang measure may be helpful in evaluating a software unit's value in terms of how much useful function it provides, although there is little evidence in the literature of such application. The use of Bang measure could apply when reengineering (either complete or piecewise) is being considered, as discussed in Maintenance of Operational Systems— an Overview, pg. 237
- *Feature points.* Adds changes to improve applicability to systems with significant internal processing (e.g., operating systems, communications systems). This allows accounting for functions not readily perceivable by the user, but essential for proper operation.

Usage Considerations

There is a very large user community for function points; IFPUG has more than 1200 member companies, and they offer assistance in establishing a FPA program. The standard practices for counting and using function points are found in the IFPUG Counting Practices Manual [IFPUG 96]. Without some standardization of how the function points are enumerated and interpreted, consistent results can be difficult to obtain. Successful application seems to depend on establishing a consistent method of counting function points and keeping records to establish baseline productivity figures for your specific systems. Function measures tend to be independent of language, coding style, and software architecture, but environmental factors such as the ratio of function points to source lines of code will vary.

The proliferation of refinements and variations of FPA noted in Technical Detail, pg. 195, has led to fragmentation. To remedy this, a Joint Technical Committee (JTC1) of the International Standards Organization (ISO) has been working since 1993 to develop ISO standards for sizing methods [Rehesaar 96]. This standardization effort is now called Functional Size Measurement.

Counting the function points needed for FPA remains largely a manual operation. This is an impediment to use. Wittig offers an approach to partial automation of function point counting [Wittig 94].

There are continuing concerns about the reliability and consistency of function point counts, such as

- whether two trained human counters will produce the same result for the same system
- the lack of inter-method reliability resulting from the variations described in Technical Detail on pg. 195

These reliability questions are addressed in a practical research effort described in Kemerer [Kemerer 93]. Siddiqee presents FPA as a good measure of productivity in a large software production environment in Lockheed Corporation [Siddiqee 93].

Any systematic FPA effort should collect the information into a database for ongoing analysis as the code is developed and/or modified.

Maturity

FPA is in use in many industrial software companies; IFPUG is large, with more than 1200 member companies, and offers many resources. As noted above, however, an ISO-level standard is still in the making.

Costs and Limitations

Currently, function point counting is a time-consuming and largely manual activity unless tools are built to assist the process. Wittig and Kemerer cite that it took more than five days to count a 4,000 function point system [Wittig 94, Kemerer 93]. However, the level of acceptance by software companies indicates that FPA is useful. Training in FPA is highly recommended; IFPUG can assist in securing training and locating FPA tools [IFPUG 96].

Alternatives

For estimation of effort, approaches based on lines of code (LOC) are an alternative. The now-classic COCOMO (constructive cost model) method and its REVIC (revised intermediate COCOMO) implementation provide a discipline for using LOC as a software size estimator [Boehm 81].

Complementary Technologies

LOC can also be used in a complementary sense as a check on results. There is also a technique called Backfiring that consists of a set of bidirectional equations for converting between function points and LOC [Jones 95]. This is reportedly useful when using sizing data from a combination of projects, some with metrics in LOC and some in function points. However, generalizing the Backfiring technique to yield a simple LOC-per-function point ratio is *not* advisable.

Index Categories

Name of technology	Function Point Analysis
Application category	Cost Estimation (AP.1.3.7)
Quality measures category	Productivity (QM.5.2)
Computing reviews category	Software Engineering Metrics (D.2.8), Software Engineering Management (D.2.9)

References and Information Sources

[Boehm 81] Boehm, Barry W. *Software Engineering Economics.* Englewood Cliffs, NJ: Prentice-Hall, 1981.

[DeMarco 82] DeMarco, Tom. *Controlling Software Projects: Management, Measurement, and Estimation.* New York, NY: Yourdon Press, 1982.

[Dreger 89] Dreger, J. Brian. *Function Point Analysis.* Englewood Cliffs, NJ: Prentice Hall, 1989.

[Heller 95] Heller, Roger. "An Introduction to Function Point Analysis," *Crosstalk, Journal of Defense Software Engineering 8*, 11 (November/December 1995): 24-26.

[IFPUG 96] The International Function Point Users' Group (IFPUG) Web site [online]. Available WWW <URL: http://www.bannister.com/ifpug/home/docs/ifpughome.html> (1996).

[Jones 95] Jones, Capers. "Backfiring: Converting Lines of Code to Function Points." *IEEE Computer 28*, 11 (November 1995): 87-8.

[Kemerer 93] Kemerer, Chris. "Reliability of Function Points Measurement: A Field Experiment." *Communications of the ACM 36*, 2 (February 1993): 85-97.

[Marciniak 94] Marciniak, John J., ed. *Encyclopedia of Software Engineering*, 518-524. New York: John Wiley & Sons, 1994.

[Rehesaar 96] Rehesaar, Hugo. "ISO/IEC Functional Size Measurement Standards," 311-318. *Proceedings of the GUFPI/IFPUG Conference on Software Measurement and Management.* Rome, Italy, February 5-9, 1996. Westerville, OH: International Function Point Users Group, 1996.

[Siddiqee 93] Siddiqee, M. Waheed. "Function Point Delivery Rates Under Various Environments: Some Actual Results," 259-264. *Proceedings of the Computer Management Group's International Conference.* San Diego, CA, December 5-10, 1993. Chicago, IL: Computer Management Group, 1993.

[Umholtz 94] Umholtz, Donald C. & Leitgeb, Arthur J. "Engineering Function Points and Tracking Systems." *Crosstalk, Journal of Defense Software Engineering 7*, 11 (November 1994): 9-14.

[Wittig 94] Wittig, G. E. & Finnie, G. R. "Software Design for the Automation of Unadjusted Function Point Counting," 613-623. *Business Process Re-Engineering Information Systems Opportunities and Challenges, IFIP TC8 Open Conference.* Gold Coast, Queensland, Australia, May 8-11, 1994. The Netherlands: IFIP, 1994.

Author Edmond VanDoren, Kaman Sciences
bvandoren-cos3@kaman.com

Last Modified 10 Jan 97

Graphic Tools for Legacy Database Migration ADVANCED

Purpose and Origin

Graphic tools for legacy database migration are used to aid in the examination of legacy data in preparation for migration of one or more databases, often as part of a system migration or reengineering effort. They are intended to enhance *understandability* and *portability* of databases by providing easily-manipulated views of both content and structure that facilitate analysis [Selfridge 94].

Technical Detail

A graphical tool portrays a database's organization and data in graphical form. This enhances the understandability of the database(s) by allowing the analyst to assess the condition and organization of the data, including overlap and duplication of data items, in preparation for migration. This enhancement is desirable for several reasons:

1. Databases are typically complex, and may lack adequate documentation.
2. The information to be migrated may be contained in several separate databases built for different purposes.

The latter usually creates data redundancy, including multiple instances of a field, and even different representations of the same data (e.g., floating point in one place, fixed point or text in another). Important legacy information may be buried in text fields that must be found in order to capture the data's content. Bennett describes some of these problems [Bennett 95]. Legacy database migration is usually done to improve a system's *maintainability* (*modifiability*, *testability*, and/or ease of *life cycle evolution*). Database migration is typically performed as part of a larger system reengineering effort. It is a branch of database design and engineering, and requires the same set of disciplines.

Usage Considerations

A visualization tool is only part of the toolset of interest in migrating legacy data. Other tools might include the following:

- data modelers
- data entry and/or query screen translators
- report translators
- data-moving and translation utilities

A migration strategy is required to create a normalized file structure with referential integrity out of large, multiple databases. The tools must fit the environment, and the target database must be interfaced with the system and application software; this implies the need for *compatibility* with the languages used. The design of the target database can greatly affect

performance and *maintainability*; therefore the first goal of the migration effort should be to define a target schema suitable for the application.

Maturity

Major database vendors offer tools of this type; the tools are typically optimized toward their database product as the target, but they accept other databases as input. There are also independent sources of visualization tools, as well as tools produced by research efforts [Selfridge 94, Gray 94]. Database migration, when offered as a service, often uses visualization tools to facilitate understanding between customer and consultant about the migration approach, process, and results [Ning 94].

Costs and Limitations

The cost of such a tool, including training, should be nominal compared to the total cost of the target database system's software, and may even be included. However, the migration itself can be costly in time and training; experience is required for good, normalized database design.

Dependencies

A migration effort would typically be coincident with a reengineering of the software that access the data, and would be intimately tied to the approaches used to do this reengineering.

Alternatives

An alternative to migration of the database is to link existing heterogeneous databases to each other. This approach eliminates the need to migrate the data, but also retains all the structural inefficiencies of the current databases, and may degrade performance. It may also create *maintainability* problems because each old database will require a separate knowledge set, and because their platforms may be not be supportable. The approach requires writing interface software that act as gateways to the other database management systems (DBMS), file systems, and/or other existing applications. The object request broker technology (see pg. 291) exemplified by the emerging Common Object Broker Architecture (CORBA) standard (see pg. 107), as well as products offered by commercial database vendors, offer the capability to link existing heterogeneous databases. This includes the ability to associate data elements in different databases, and do JOINS across database boundaries.

Complementary Technologies

Other tools for analyzing data content and structure are available from commercial vendors and academic and research organizations. Knowledge-based approaches, for example, may have the ability to infer identity between multiple, differently-named instances of a data item. Other approaches such as these can compliment the use of graphical analyzers. Migration is typically done in the context of open systems (see pg. 135), which implies a large number of technologies that would be helpful together.

Index Categories

Name of technology	Graphic Tools for Legacy Database Migration
Application category	Database Design (AP.1.3.2), Reengineering (AP.1.9.5)
Quality measures category	Understandability (QM.3.2), Maintainability (QM.3.1), Testability (QM.1.4.1), Compatibility (QM.4.1.1), Throughput (QM.2.2.3)
Computing reviews category	Database Management - Logical Design (H.2.1)

References and Information Sources

[Bennett 95] Bennett, K. "Legacy Systems: Coping With Stress." *IEEE Software 12,* 1 (January 1995): 19-23.

[Gray 94] Gray, W. A.; Wikramanayake, G. N.; & Fiddian, N. J. "Assisting Legacy Database Migration," 5/1-3. *IEE Colloquium: Legacy Information System— Barriers to Business Process Re-Engineering* (1994/246). London, UK, December 13, 1994. London, UK: IEE, 1994.

[Ning 94] Ning, Jim Q.; Engberts, Andre; & Kozaczynski, W. "Automated Support for Legacy Code Understanding." *Communications of the ACM 37,* 5 (May 1994): 50-57.

[Selfridge 94] Selfridge, Peter G. & Heineman, George T. "Graphical Support for Code-Level Software Understanding," 114-24. *Ninth Knowledge-Based Software Engineering Conference.* Monterey, CA, September 1994. Los Alamitos, CA: IEEE Computer Society Press, 1994.

Author

Edmond VanDoren, Kaman Sciences
bvandoren-cos3@kaman.com

Last Modified 10 Jan 97

Graphical User Interface Builders

ADVANCED

Purpose and Origin

Graphical user interface (GUI) builders are software engineering tools developed to increase the productivity of user interface (UI) development teams, and to lower the cost of UI code in both the development and maintenance phases. One study found that an average of 48% of application code is devoted to the UI, and 50% of the development time required for the entire application is devoted to the UI portion [Myers 95]. Use of GUI builders can significantly reduce these numbers. For example, the MacApp system from Apple has been reported to reduce development time by a factor of four. Another study found that an application using a popular GUI tool wrote 83% fewer lines of code and took one-half the time compared to applications written without GUI tools [Myers 95]. Original GUI research was conducted at the Stanford Research Institute, Xerox Palo Alto Research Center, and Massachusetts Institute of Technology in the 1970s [Myers 95].

Technical Detail

A GUI development tool simplifies the coding of complex UI applications by providing the developer with building blocks (or widgets) of UI components. These building blocks are manipulated by the developer into a cohesive UI allowing a smaller workforce to develop larger amounts of user interface software in shorter time periods. A GUI builder enhances *usability* by providing a development team with a prototyping capability so that proposed UI changes can be rapidly demonstrated to the end user to secure requirements validation and acceptance. This aspect can decrease the turnaround time for making UI changes in the Operations and Maintenance (O&M) phase, which enhances *maintainability* as well.

GUI development tools can be broadly categorized into two types:

- Interface Development Tools (IDTs)
- User Interface Management Systems (UIMSs)

IDTs are used for building the interface itself, but nothing more. By contrast, UIMSs extend the functionality of IDTs to include application development (code generation tools) or scripting tools. A UIMS also allows the developer to specify the behavior of an application with respect to the interface. These two types of GUI builders permit the interactive creation of the front-end GUI using a palette of widgets, a widget attribute specification form, a menu hierarchy (menu tree structure), a tool bar, and a view of the form. The UIMS adds the benefits of code generation tools, which can greatly increase the productivity of the GUI development staff. After the front-end is created by a UIMS, a code generator is used to pro-

duce C/C++ code, Motif User Interface Language (UIL) code, Ada code or some combination of C, Ada, and UIL.

Usage Considerations

GUI builders are useful for development of complex user interfaces because they increase software development speed by providing tools to lay out screens graphically and automatically generate interface code. Additionally, in applications that are susceptible to continuing user interface change such as command and control applications, the use of GUI builders greatly increases the ability to add/modify user interface functionality in minimal time to support mission changes or new requirements.

This technology works best when used in new development and reengineering. To take full advantage of the benefits of using GUI builders, the most desirable software architecture would be one that ensures the user interface software is isolated on a single layer as opposed to being embedded within several different software components. This isolation simplifies the UI portion of the software, thus making changes during development easier as well as enhancing future maintainability and evolvability.

Maturity

From the early GUI research started in the 1970s, GUI builder tools have grown into an estimated $1.2 billion business [Myers 95]. Today there are literally hundreds of GUI builders on the market supporting platforms ranging from UNIX to DOS. Virtually all new commercial and government applications use some form of UI builder tool. GUI builders have been successfully used on legacy systems when large changes or UI redesigns were applied to the user interface portion of the software [Myers 95].

Costs and Limitations

This technology requires workstations or PCs dedicated to support the development, rapid prototype, and validation of user interfaces. The most widely used GUI builders on the market today require minimal learning time for C and C++ trained developers. These packages come with appropriate training materials, online help features, and vendor-supplied help lines which help make the developers productive in minimal time. There are few formal training costs associated with the use of GUI builders; however an organization would be well advised to provide internal training focusing on standardized approaches and techniques similar to design and coding standards for source code.

The prime costs with GUI builders are the initial license fees, annual maintenance agreements, and the cost of the workstations. In the UNIX environment, typical license costs for full UIMS GUI builders are in the range of $5k to $7.5k per single user license. For Windows or Macintosh

environments, the costs range from $300 to $600 per user license. The maintenance agreements are key to keeping each GUI builder updated with vendor corrections and upgrades.

Dependencies

GUI development tools employ window managers as the foundation upon which a user interface can be built. A window manager allows the user to display, alter, and interact with more than one window at a time. The window manager's primary responsibility is to keep track of all aspects of each of the windows being displayed. In terms of numbers of applications in use, the two most popular window managers are Open Windows and Motif from Open Software Foundation (OSF) [OSF 96].

Alternatives

UI software can be developed without the use of GUI builders by using the features of window managers. For example, developers can use the X Windows based Motif (from OSF) and its rich set of widgets and features to design and implement UIs. This may be desirable for smaller, less complex UI applications for which the developer does not require the assistance (and extra cost) of GUI builders.

Index Categories

Name of technology	Graphical User Interface Builders
Application category	Interfaces Design (AP.1.3.3), Code (AP.1.4.2), Reapply Software Life Cycle (AP.1.9.3), Reengineering (AP.1.9.5)
Quality measures category	Usability (QM.2.3), Maintainability (QM.3.1)
Computing reviews category	Software Engineering Tools and Techniques (D.2.2), User Interfaces (H.1.2)

References and Information Sources

[Myers 95] Myers, Brad A. "User Interface Software Tools." *ACM Transactions on Computer-Human Interaction 2,* 1 (March 1995): 64-108.

[OSF 96] OSF Home Page [online]. Available WWW <URL: http://www.osf.org> (1996).

Author Mike Bray, Lockheed-Martin Ground Systems
michael.w.bray@den.mmc.com

External Reviewer(s) Brian Gallagher, SEI

Last Modified 10 Jan 97

Halstead Complexity Measures

ADVANCED

Note

We recommend that Maintainability Index Technique for Measuring Program Maintainability, pg. 231, be read concurrently with this technology description, It illustrates a specific application of Halstead complexity to quantify the maintainability of software.

Purpose and Origin

Halstead complexity measurement was developed to measure a program module's *complexity* directly from source code, with emphasis on computational complexity. The measures were developed by the late Maurice Halstead as a means of determining a quantitative measure of complexity directly from the operators and operands in the module [Halstead 77]. Among the earliest software metrics, they are strong indicators of code complexity. Because they are applied to code, they are most often used as a maintenance metric. There are widely differing opinions on the worth of Halstead measures, ranging from "convoluted... [and] unreliable" [Jones 94] to "among the strongest measures of maintainability" [Oman 92]. The material in this technology description is largely based on the empirical evidence found in the Maintainability Index work, but there is evidence that Halstead measures are also useful during development, to assess code quality in computationally-dense applications.

Technical Detail

The Halstead measures are based on four scalar numbers derived directly from a program's source code:

n1 = the number of distinct operators
n2 = the number of distinct operands
N1 = the total number of operators
N2 = the total number of operands

From these numbers, five measures are derived:

Measure	Symbol	Formula
Program length	N	N= N1 + N2
Program vocabulary	n	n= n1 + n2
Volume	V	V= N * (LOG2 n)
Difficulty	D	D= (n1/2) * (N2/n2)
Effort	E	E= D * V

These measures are simple to calculate once the rules for identifying operators and operands have been determined (Szulewski notes that establishing these rules can be quite difficult [Szulewski 84]). The extraction

of the component numbers from code requires a language-sensitive scanner, which is a reasonably simple program for most languages. Oman describes a tool for use in determining maintainability which, for Pascal and C, computes the following [Oman 91]:

V for each module; and

V(g), the average Halstead volume per module for a system of programs

For Pascal alone, the following are also computed:

E for each module; and

E(g), the average Halstead volume per module for a system of programs

Usage Considerations

Applicability. The Halstead measures are applicable to operational systems and to development efforts once the code has been written. Because maintainability should be a concern during development, the Halstead measures should be considered for use during code development to follow complexity trends. A significant complexity measure increase during testing may be the sign of a brittle or high-risk module. Halstead measures have been criticized for a variety of reasons, among them the claim that they are a weak measure because they measure lexical and/or textual complexity rather than the structural or logic flow complexity exemplified by cyclomatic complexity measures (see pg. 145). However, they have been shown to be a very strong component of the Maintainability Index measurement of maintainability (see pg. 231). In particular, the complexity of code with a high ratio of calculational logic to branch logic may be more accurately assessed by Halstead measures than by cyclomatic complexity (see pg. 145), which measures structural complexity.

Relation to other complexity measures. Marciniak describes all of the commonly-known software complexity measures and puts them in a common framework [Marciniak 94]. This is helpful background for any complexity measurement effort. Most measurement programs benefit from using several measures, at least initially; discarding those that do not suit the specific environment; and combining those that work (see Complementary Technologies on pg. 211). This is illustrated by Maintainability Index Technique for Measuring Program Maintainability, pg. 231, which describes the use of Halstead measures in combination with other complexity measures. When used in this context, the problems with establishing rules for identifying the elements to be counted are eliminated.

Maturity

Halstead measures were introduced in 1977 and have been used and experimented with extensively since that time. They are one of the oldest

measures of program complexity. Because of the criticisms mentioned above, they have seen limited use. However, their properties are well-known and, in the context explained in Usage Considerations, pg. 210, they can be quite useful.

Costs and Limitations

The algorithms are free; the tool described in Technical Detail, pg. 209, contains Halstead scanners for Pascal and C, and some commercially-available CASE toolsets include the Halstead measures as part of their metric set. For languages not supported, standalone scanners can probably be written inexpensively, and the results can be exported to a spreadsheet or database to do the calculations and store the results for use as metrics. It should be noted that difficulties sometimes arise in uniquely identifying operators and operands. Consistency is important. Szulewski discusses this, defines consistent counting techniques for Ada, and points to other sources of counting techniques for some other languages [Szulewski 84]. Adding Halstead measures to an existing maintenance environment's metrics collection effort and then applying them to the software maintenance process will require not only the code scanner, but a collection system that feeds the resulting data to the metrics effort. Halstead measures may not be sufficient by themselves as software metrics (see Complementary Technologies).

Alternatives

Common practice today is to combine measures to suit the specific program environment. Most measures are amenable for use in combination with others (although some overlap). Thus, many alternative measures are to some degree complementary. Oman presents a very comprehensive list of code metrics that are found in maintainability analysis work, and orders them by degree of influence on the maintainability measure being developed in that effort [Oman 94]. Some examples are (all are averages across the set of programs being measured)

- lines of code per module
- lines of comments per module
- variable span per module
- lines of data declarations per module

Complementary Technologies

Cyclomatic complexity and its associated complexity measures (see pg. 145) measure the structural complexity of a program. Maintainability Index Technique for Measuring Program Maintainability, pg. 231, combines cyclomatic complexity with Halstead measures to produce a practical measure of maintainability.

Function point measures (see pg. 195) provide a measure of functionality, with some significant limitations (at least in the basic function point

enumeration method); the variant called engineering function points adds measurement of mathematical functionality that may complement Halstead measures.

Lines-of-code (LOC) metrics offer a gross measure of code, but do not measure content well. However, LOC in combination with Halstead measures may help relate program size to functionality.

Index Categories

Name of technology	Halstead Complexity Measures
Application category	Code (AP.1.4.2), Debugger (AP.1.4.2.4), Test (AP.1.4.3), Unit Testing (AP.1.4.3.4), Component Testing (AP.1.4.3.5), Reapply Software Life Cycle (AP.1.9.3), Reengineering (AP.1.9.5)
Quality measures category	Maintainability (QM.3.1), Testability (QM.1.4.1), Understandability (QM.3.2), Complexity (QM.3.2.1)
Computing reviews category	Software Engineering Distribution and Maintenance (D.2.7), Software Engineering Metrics (D.2.8), Complexity Classes (F.1.3), Tradeoffs Among Complexity Measures (F.2.3)

References and Information Sources

[Halstead 77] Halstead, Maurice H. *Elements of Software Science, Operating, and Programming Systems Series* Volume 7. New York, NY: Elsevier, 1977.

[Jones 94] Jones, Capers. "Software Metrics: Good, Bad, and Missing." *Computer 27,* 9 (September 1994): 98-100.

[Marciniak 94] Marciniak, John J., ed. *Encyclopedia of Software Engineering,* 131-165. New York: John Wiley & Sons, 1994.

[Oman 91] Oman, P. *HP-MAS: A Tool for Software Maintainability, Software Engineering* (#91-08-TR). Moscow, ID: Test Laboratory, University of Idaho, 1991.

[Oman 94] Oman, P. & Hagemeister, J. "Constructing and Testing of Polynomials Predicting Software Maintainability." *Journal of Systems and Software 24,* 3 (March 1994): 251-266.

[Szulewski 84] Szulewski, Paul, et al. *Automating Software Design Metrics* (RADC-TR-84-27). Rome, NY: Rome Air Development Center, 1984.

Author Edmond VanDoren, Kaman Sciences
bvandoren-cos3@kaman.com

Last Modified 10 Jan 97

Hybrid Automata

ADVANCED

Purpose and Origin

Hybrid automata form the basis for a specification and design technique for use in software support tools [Henzinger 94]. They were developed by Thomas Henzinger to broaden formal specifications to include continuous variables, such as response time and distance — that describe a system's operating environment.

Technical Detail

Hybrid automata increase the completeness of specifications and the fidelity of models by allowing continuous properties of the operating environment to be specified and modeled directly. Hybrid automata are extensions of finite state automata to continuous quantities. Finite state automata provide a mathematical foundation for reasoning about systems in terms of their discrete properties. In hybrid automata, state transitions may be triggered by functions on continuous variables. Any linear continuous property of a system can be specified and modeled using this technique. It is not clear whether hybrid automata can be usefully extended to nonlinear continuous variables.

Usage Considerations

Hybrid automata are useful for developing systems that must interact in a substantial way with the physical world. Response time, as required in command and control, avionics, and air traffic control, is an example of such interaction. Because the resulting models are more faithful to reality, hybrid automata will likely contribute to increased correctness and reliability. Additional work is needed to determine whether this technique is extendible to nonlinear continuous variables and scalable to large systems of linear continuous variables.

Maturity

The technique was devised around 1992 with a prototype model checker, HyTech, developed in 1994. The technique has been applied experimentally to a few cases, including verification of an industrial converter between analog and digital signals. This converter uses distributed clocks that may drift apart. The model checker automatically computes maximum clock drift so that the converter works correctly.

Costs and Limitations

Adaptation of this technique requires knowledge of discrete mathematics at the level of automata theory and continuous mathematics at the level of differential equations.

Dependencies

Hybrid automata are enablers for technologies that check the consistency of requirements for contiguous properties.

Alternatives

Other approaches to capturing and processing continuous properties of a system's operating environment have been stochastic methods, probabilistic automata, and dynamic simulation.

Complementary Technologies

Model checking is a useful approach for verifying that hybrid automata meet a specific requirement.

Index Categories

Name of technology	Hybrid Automata
Application category	Detailed Design (AP.1.3.5)
Quality measures category	Completeness (QM.1.3.1), Fidelity (QM.2.4), Correctness (QM.1.3), Reliability (QM.2.1.2)
Computing reviews category	Models of Computation (F.1.1)

References and Information Sources

[Henzinger 94] Henzinger, T.A. & Ho, P. "HYTECH: The Cornell HYbrid TECHnology Tool," 265-93. *Proceedings of the 1994 Workshop on Hybrid Systems and Autonomous Control.* Berlin, Germany, October 28-30, 1994. Berlin, Germany: Springer-Verlag, 1995.

Author

David Fisher, SEI
dfisher@sei.cmu.edu

Major David Luginbuhl, Air Force Office of Scientific Research
david.luginbuhl@afosr.af.mil

External Reviewer(s)

Tom Henzinger, Assistant Professor Electrical Engineering and Computer Sciences, University of California at Berkeley

Last Modified

10 Jan 97

Intrusion Detection

ADVANCED

Note

We recommend Computer System Security— an Overview, pg. 129, as prerequisite reading for this technology description.

Purpose and Origin

In the mid to late 1960s, as time sharing systems emerged, controlling access to computer resources became a concern. In the 1970s, the Department of Defense (DoD) Ware Report pointed out the need for computer security [Ware 79]. In the mid to late 1970s, a number of systems were designed and implemented using security kernel architectures. In the late 1970s, Tiger Teams began to evaluate the security of various systems. In 1983, the *Department of Defense Trusted Computer System Evaluation Criteria* — the "orange book" — was published and provided a set of criteria for evaluating computer security control effectiveness [DoD 85]. Research in this area continued through the 1980s, but many facets of computer security control remained a largely manual process. For example, the Internet Worm program of 1988 — which infected thousands of machines and disrupted normal activities for several days— was detected primarily through manual means [Spafford 88]. Today, there are primarily four approaches to achieving a secure computing environment [Kemmerer 94]:

1. the use of special procedures — such as password selection and use, access control, and manual review of output products— for working with a system
2. the inclusion of additional functions or mechanisms in the system
3. the use of assurance techniques — such as penetration analysis, formal specification and verification, and covert channel analysis — to increase one's confidence in the security of a system
4. the use of intrusion detection systems (IDSs)

The fourth approach, intrusion detection, is an emerging technology that seeks to automate the detection and elimination of intrusions. IDSs seek to increase the *security* and hence the *availability, integrity*, and *confidentiality* of computer systems by eliminating unauthorized system/data access.

Technical Detail

Intrusion detection systems (IDSs) are predicated on the assumption that an intruder can be detected through an examination of various parameters such as network traffic, CPU utilization, I/O utilization, user location, and various file activities [Lunt 93]. System monitors or daemons convert observed parameters into chronologically sorted records of system activ-

ities. Called "audit trails," these records are analyzed by IDSs for unusual or suspect behavior. IDS approaches include

- rule-based intrusion detection (see pg. 331)
- statistical-based intrusion detection (see pg. 357)

IDSs designed to protect networks typically monitor network activity, while IDSs designed for single hosts typically monitor operating system activity.

Usage Considerations

Although IDSs are likely to increase the security of computer systems, the collection and processing of audit data will degrade system performance. Note that an IDS can be used to augment crypto-based security systems— which cannot defend against cracked passwords or lost or stolen keys— and to detect the abuse of privileges by authorized users [Mukherjee 94]. User authentication systems can be used to augment IDS systems.

Maturity

Prototypes of several intrusion detection systems have been developed, and some intrusion detection systems have been deployed on an experimental basis in operational systems. At least one network-based IDS — the Network Security Monitor (NSM) — successfully detected an attack in which an intruder exploited known security flaws to gain access to systems distributed over seven sites, three states, and two countries [Mukherjee 94]. However, additional work is required to determine appropriate levels of auditing, to strengthen the representation of intrusion attempts, and to extend the concept of intrusion detection to arbitrarily large networks [Lunt 93, Mukherjee 94].

Costs and Limitations

Audit trail analysis can be conducted either offline (after the fact) or in real time. Although offline analysis permits greater depth of coverage while shifting the processing of audit information to non-peak times, it can only detect intrusions after the fact. Real-time IDSs can potentially catch intrusion attempts before the system state is compromised, but real-time IDSs must run concurrently with other system applications and will therefore negatively affect *throughput*. In addition to the costs associated with creating and analyzing audit trails, IDS systems cannot detect all intrusion attempts, primarily because only known intrusion scenarios can be represented. An intrusion attempt made using a scenario not represented by an IDS system may be successful, and some intrusion attempts have succeeded in either turning off the audit daemon or in modifying the audit data prior to its being processed by an IDS.

Although most IDSs are designed to support multiple operating systems, audit data collected by monitoring operating system activity will be oper-

ating system specific [Mukherjee 94]; this type of data may therefore need to be converted into a standard form before it can be processed by an IDS.

For these reasons, many IDS systems are designed as assistants to human computer security monitors.

Dependencies

System or network auditing tools and techniques are necessary enablers for this technology. Depending on the type of IDS, expert systems technology may also be needed.

Index Categories

Name of technology	Intrusion Detection
Application category	System Security (AP.2.4.3)
Quality measures category	Security (QM.2.1.5)
Computing reviews category	Operating Systems Security and Protection (D.4.6), Computer-Communication Networks Security and Protection (C.2.0), Security and Protection (K.6.5)

References and Information Sources

[DoD 85] *Department of Defense Trusted Computer System Evaluation Criteria*, DoD standard DoD 5200.28-STD [online]. Available WWW <URL: http://www.v-one.com/newpages/obook.html> (1985).

[Kemmerer 94] Kemmerer, Richard A. "Computer Security," 1153-1164. *Encyclopedia of Software Engineering.* New York, NY: John Wiley and Sons, 1994.

[Lunt 93] Lunt, Teresa F. "A Survey of Intrusion Detection Techniques." *Computers and Security 12,* 4 (June 1993): 405-418.

[Mukherjee 94] Mukherjee, Biswanath, L.; Heberlein, Todd; & Levitt, Karl N. "Network Intrusion Detection." *IEEE Network 8,* 3 (May/June 1994): 26-41.

[Smaha 88] Smaha, Stephen E. "Haystack: An Intrusion Detection System," 37-44. *Proceedings of the Fourth Aerospace Computer Security Applications Conference.* Orlando, Florida, December 12-16, 1988. Washington, DC: IEEE Computer Society Press, 1989.

[Sundaram 96] Sundaram, Aurobindo. *An Introduction to Intrusion Detection* [online]. Available WWW <URL: http://www.acm.org/crossroads/xrds2-4/xrds2-4.html> (1996).

[Spafford 88] Spafford, Eugene H. *The Internet Worm Program: An Analysis* (CSD-TR-823). West Lafayette, IN: Purdue University, 1988.

[Ware 79] Ware, W. H. *Security Controls for Computer Systems: Report of Defense Science Board, Task Force on Computer Security.* Santa Monica, CA: The Rand Corporation, 1979.

Author Mark Gerken, Rome Laboratory
gerken@ai.rl.af.mil

Last Modified 10 Jan 97

Java

ADVANCED

Purpose and Origin

Java is a hardware- and implementation-independent programming language originally developed by a team of engineers headed by James Gosling at Sun Microsystems (development began in 1991). Java addresses many of the issues of software distribution over a network, including *interoperability*, *security*, and *portability*.

Java is an interpreted language that uses a virtual machine, or runtime environment, on the client computer that can be executed on any machine to which the interpreter has been ported. Java virtual machines were developed initially for use as part of World Wide Web (WWW) browsers but are continuing to be developed for a wider range of platforms such as operating systems and telephones [Hamilton 96].

Java has potential for revolutionizing the WWW [Sun 96b]. Java enables a programmer to extend Internet browsers by enabling Java programs, called "applets," to be embedded on a Web page [Singleton 1996]. Embedding Java applets into Web pages provides a developer the *flexibility* to develop a more sophisticated user interface on a Web page [Yourdon 96]. Java applets can provide a full range of event-driven pop-up windows and graphical user interface (GUI) widgets (see pg. 205), which can support a variety of applications such as animations, simulations, teaching tools, and spreadsheets [van Hoff 95].

Technical Detail

Java programs provide portability and a measure of *security*. Java programs are compiled into a byte-code format (a low-level, pseudo-machine language) called J-code that can be executed on many platforms without recompilation. The J-code is interpreted at runtime and since the J-code is consistent, interpreters have been written for a significant population of processor architectures, thus allowing for portability across various processor architectures and hardware. Java is designed to allow Applets to be downloaded without introducing viruses or misbehaved code. This intent has been somewhat realized, although several weaknesses in the security design have been found [Sun 96a].

The Java virtual machine is typically installed on a user's machine as either part of a Web browser or as part of the operating system. Java provides flexibility in that it allows a user to download the specific functionality the user wants at the moment the user requests that functionality. This is a paradigm shift from the normal model, which requires the entire suite of possible functionality to be installed onto a user's platform prior to execution [Yourdon 96].

Java applications are arguably more robust than C or C++ applications. Java has features, like C++-style exceptions, runtime type checking, and automatic garbage collection and memory management, that allow the writing of robust code that can recover from runtime errors [Hamilton 96]. Several methods/techniques for garbage collection are provided. Java garbage collection aids the programmer by eliminating the need for a user to explicitly free memory, thus eliminating a whole class of memory bugs and the tools designed to check for them [Hamilton 96]. In addition, Java is object-oriented (see pg. 287); it provides support for a single-inheritance class hierarchy, and an efficient implementation of method invocations for both virtual and static methods.

Usage Considerations

Java is intended for use in the development of large distributed systems. Java specifies a core set of application programming interfaces (APIs)(see pg. 79)— required in all Java implementations— and an extended set of APIs covering a much broader set of functionality. The core set of APIs include interfaces for

- basic language types
- file and stream I/O
- network I/O
- container and utility classes
- abstract windowing toolkit

The extended set of APIs includes interfaces for 2D-rendering and 2D-animation; a 3D-programming model; telephony, time-critical audio, video, and MIDI data; network and systems management; electronic commerce; and encryption and authentication [Hamilton 96].

The Java syntax for expressions and statements are almost identical to ANSI C, thus making the language easy to learn for C or C++ programmers. Because Java is a programming language, it requires a higher skill level for content developers than hypertext markup language (HTML). Programmers need to learn the Java standard library, which contains objects and methods for opening sockets, implementing the HTTP protocol, creating threads, writing to the display, and building a user interface. Java provides mechanisms for interfacing with other programming languages such as C and existing libraries such as Xlib, Motif, or legacy database software.

The code generated by Java is small— the basic interpreter and class support are about 40Kbytes; thread support and basic standard libraries are another 175K.

Performance is a major consideration when deciding to use Java. In some comparisons, the Java interpreter was about 10-15 times slower than compiled C or C++, which should be acceptable for most applications [van Hoff 95]. Another comparison indicates that C++ is 30 times faster than Java [Yourdon 96]. If performance must be increased, it is possible and straightforward to convert the machine-independent bytecodes to machine instructions. The performance implications of the Java garbage collector should also be considered, especially for real-time applications. Note that performance has been improving; as of September 1996, code from a just in time (JIT) compiler supporting Java was benchmarked at 0.8 of an optimizing C++ compiler [Coffee 96].

A number of items should be considered if migrating from C or C++ to Java, including the following:

- Java is totally object-oriented, thus everything must be done via a method invocation.
- Java has no pointers or parameterized types.
- Java supports multithreading and garbage collection [Aitken 96].

Maturity

Java was made available to the general public in May 1995, and has enjoyed unprecedented transition into practice:

- As of September 1996, Java has been licensed by over 35 vendors, and is supported by the leading operating systems, including Windows, Macintosh, and many variants of UNIX [Elmer-Dewitt 96].
- More than 5,000 applets have appeared on the Internet [Hamilton 96].
- Java support has been incorporated into all leading Web browsers (e.g., Netscape Navigator v3.0 and Microsoft Internet Explorer v3.0).
- A survey in May 1996 found that 62% of large companies already use Java for some development; another 14% expect to start using Java by the end of 1996. Over 40% of the companies say Java will "become strategic" within a year [Wilder 96].

Early implementations of Java contained bugs relating to security [Sun 96a]. These bugs were fixed for the Java Development Kit version 1.02 and Netscape Navigator 3.0b4 [Hamilton 96]. Felten examines Java security in detail [Felten 96].

One of the first, and possibly largest, mission-critical applications written in Java is TWSNet, a shipment tracking and processing application for CSX Corp. TWSNet began testing of its initial phase in April 1996 after 90 days of development. Other existing applications of Java include a stock trading service (Lombard Institutional Brokerage) and semiconduc-

tor product catalogs (National Semiconductor, Hitachi America, and Philips Electronics) [Wilder 96].

Development environments that support Java programming started to appear on the market in 1996. The environments support creation of the user interface but stop short of providing full CASE (computer aided software engineering) tool support for Java developers [Yourdon 96].

Costs and Limitations

Java and the source for the Java interpreter are freely available for non-commercial use. Some restrictions exist for incorporating Java into commercial products. Sun Microsystems licenses Java to hardware and software companies that are developing products to run the Java virtual machine and execute Java code. Developers, however, can write Java code without a license. A complete Java Development Kit, including a Java compiler, can be downloaded for free [Sun 96c, Hamilton 96].

Yourdon discusses the potential impact of Java on the cost of software applications in the future— purchased software packages could be replaced with transaction-oriented rental of Java applets attached to Web pages [Yourdon 96].

Alternatives

Alternatives include C++,C, PERL, TCL, and Smalltalk.

Complementary Technologies

Cross-compilers are being developed for Ada95 (see pg. 67).

Index Categories

Name of technology	Java
Application category	Distributed Computing (AP.2.1.2), Application Program Interfaces (AP.2.7), Programming Language (AP.1.4.2.1), Compiler (AP.1.4.2.3)
Quality measures category	Maintainability (QM.3.1), Interoperability (QM.4.1), Portability (QM.4.2), Complexity (QM.3.2.1), Trustworthiness (QM.2.1.4)
Computing reviews category	Programming Languages (D.3), Distributed Systems (C.2.4)

References and Information Sources

[Aitken 96] Aitken, G. "Moving from C++ to Java." *Dr. Dobb's Journal 21*, 3 (March 1996): 52-56.

[Carlson 95] Carlson, Bob. "A Jolt of Java Could Shake up the Computing Community." *Computer Magazine 28,* 11 (November 1995): 81-2.

[Elmer-Dewitt 96] Elmer-Dewitt, Philip. "Why Java is Hot." *Time 147,* 4 (January 22, 1996): 58-60.

[Felten 96] Felten, E.W.; Dean, D.; & Wallach, D.S. "Java Security: From Hot Java to Netscape and Beyond," 190-200. *Proceedings of the 1996 IEEE Symposium on Security and Privacy.* Oakland, CA, May 6-8, 1996. Los Alamitos, CA: IEEE Computer Society Press, 1996.

[Hamilton 96] Hamilton, Marc. "Java and the Shift to Net-Centric Computing." *Computer 29*, 8 (August 1996): 31-39.

[Singleton 96] Singleton, Andrew. "Wired on the Web." *BYTE 21,* 1 (January 1996): 58-61.

[Coffee 96] Coffee, Peter. "I'm Not Supposed To Make up My Mind." *PC Week 13,* 34 (August 26, 1996): 16. Also available [online] WWW <URL: http://www. pcweek.com/archive/1334/pcwk0089.htm> (1996).

[Sun 96a] *Frequently Asked Questions— Applet Security* [online]. Available WWW <URL: http://java.sun.com/sfaq> (1996).

[Sun 96b] *Java Computing in the Enterprise. Strategic Overview: Java* [online]. Available WWW <URL: http://www.sun.com/javacomputing> (1996).

[Sun 96c] Sun Java Web site [online]. Available WWW <URL: http://java.sun.com> (1996).

[van Hoff 95] van Hoff, A. "Java and Internet Programming." *Dr. Dobb's Journal 20,* 8 (August 1995): 56-61, 101-2.

[Wilder 96] Wilder, Clinton. "Java in Gear." *Informationweek* 592 (August 12, 1996): 14-16.

[Yourdon 96] Yourdon, Edward. "Java, the Web, and Software Development." *Computer 29*, 8 (August 1996): 25-30.

Author Cory Vondrak, TRW, Redondo Beach, CA

External Reviewer(s) Archie Andrews, SEI
Scott Tilley, SEI

Last Modified 10 Jan 97

Mainframe Server Software Architectures — COMPLETE

Note

We recommend Client/Server Software Architectures, pg. 101, as prerequisite reading for this technology description.

Purpose and Origin

Since 1994 mainframes have been combined with distributed architectures to provide massive storage and to improve system security, *flexibility, scalability*, and *reusability* in the client/server design. In a mainframe server software architecture, mainframes are integrated as servers and data warehouses in a client/server environment. Additionally, mainframes still excel at simple transaction-oriented data processing to automate repetitive business tasks such as accounts receivable, accounts payable, general ledger, credit account management, and payroll. Siwolp and Edelstein provide details on mainframe server software architectures see [Siwolp 95, Edelstein 94].

Technical Detail

While client/server systems (see pg. 101) are suited for rapid application deployment and distributed processing, mainframes are efficient at online transactional processing, mass storage, centralized software distribution, and data warehousing [Data 96]. Data warehousing is information (usually in summary form) extracted from an operational database by data mining (drilling down into the information through a series of related queries). The purpose of data warehousing and data mining is to provide executive decision makers with data analysis information (such as trends and correlated results) to make and improve business decisions.

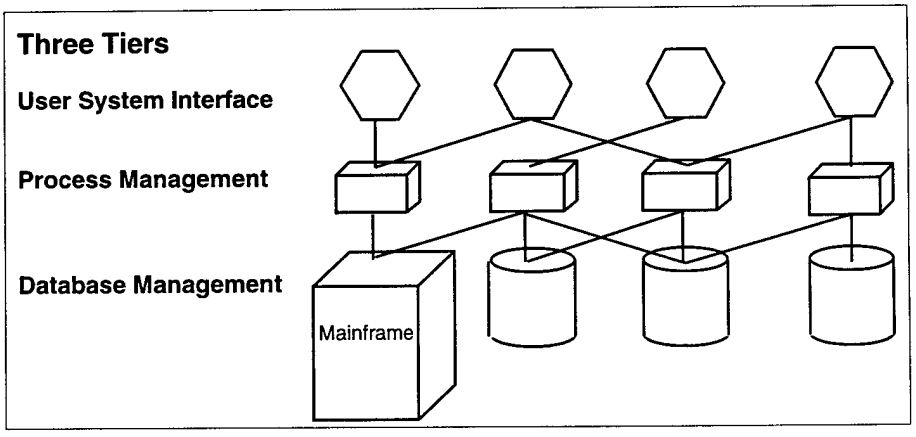


Figure 14: Using a Mainframe in a Three Tier Client/Server Architecture

Figure 14 shows a mainframe in a three tier client/server architecture. The combination of mainframe horsepower as a server in a client/server distributed architecture results in a very effective and efficient system.

Mainframe vendors are now providing standard communications and programming interfaces that make it easy to integrate mainframes as servers in a client/server architecture. Using mainframes as servers in a client/server distributed architecture provides a more modular system design, and provides the benefits of the client/server technology.

Using mainframes as servers in a client/server architecture also enables the distribution of workload between major data centers and provides disaster protection and recovery by backing up large volumes of data at disparate locations. The current model favors "thin" clients (contains primarily user interface services) with very powerful servers that do most of the extensive application and data processing, such as in a two tier architecture (see pg. 381). In a three tier client/server architecture (see pg. 367), process management (business rule execution) could be off-loaded to another server.

Usage Considerations

Mainframes are preferred for big batch jobs and storing massive amounts of vital data. They are mainly used in the banking industry, public utility systems, and for information services. Mainframes also have tools for monitoring performance of the entire system, including networks and applications not available today on UNIX servers [Siwolp 95].

New mainframes are providing parallel systems (unlike older bipolar machines) and use complementary metal-oxide semiconductor (CMOS) microprocessors, rather than emitter-coupler logic (ECL) processors. Because CMOS processors are packed more densely than ECL microprocessors, mainframes can be built much smaller and are not so power-hungry. They can also be cooled with air instead of water [Siwolp 95].

While it appeared in the early 1990s that mainframes were being replaced by client/server architectures, they are making a comeback. Some mainframe vendors have seen as much as a 66% jump in mainframe shipments in 1995 due to the new mainframe server software architecture [Siwolp 95].

Given the cost of a mainframe compared to other servers, UNIX workstations and personal computers (PCs), it is not likely that mainframes would replace all other servers in a distributed two or three tier client/server architecture.

Maturity

Mainframe technology has been well known for decades. The new improved models have been fielded since 1994. The new mainframe server software architecture provides the distributed client/server design with massive storage and improved security capability. New technologies of data warehousing and data mining data allow extraction of information

from the operational mainframe server's massive storage to provide businesses with timely data to improve overall business effectiveness. For example, stores such as Wal-Mart found that by placing certain products in close proximity within the store, both products sold at higher rates than when not collocated.[1]

Costs and Limitations

By themselves, mainframes are not appropriate mechanisms to support graphical user interfaces. Nor can they easily accommodate increases in the number of user applications or rapidly changing user needs [Edelstein 94].

Alternatives

Using a client/server architecture without a mainframe server is a possible alternative. When requirements for high volume (greater than 50 gigabit), batch type processing, security, and mass storage are minimal, three tier (see pg. 367) or two tier architectures (see pg. 381) without a mainframe server may be viable alternatives. Other possible alternatives to using mainframes in a client/server distributed environment are using parallel processing software architecture or using a database machine.

Complementary Technologies

A complementary technology to mainframe server software architectures is open systems (see pg. 135). This is because movement in the industry towards interoperable heterogeneous software programs and operating systems will continue to increase reuse of mainframe technology and provide potentially new applications for mainframe capabilities.

Index Categories

Name of technology	Mainframe Server Software Architectures
Application category	Client/Server (AP.2.1.2.1)
Quality measures category	Maintainability (QM.3.1), Scalability (QM.4.3), Reusability (QM.4.4)
Computing reviews category	Distributed Systems (C.2.4)

References and Information Sources

[Data 96] *Data Warehousing* [online]. Available WWW <URL: http://www-db.stanford.edu/warehousing/publications.html> and <URL: http://www-db.stanford.edu/warehousing/warehouse.html> (1996).

1. Source: Stodder, David. Open Session Very Large Data Base (VLDB) Summit. New Orleans, LA 23-26 April, 1995.

[Edelstein 94] Edelstein, Herb. "Unraveling Client/Server Architecture." *DBMS 7,* 5 (May 1994): 34(7).

[Siwolp 95] Siwolp, Sana. "Not Your Father's Mainframe." *Information Week* 546 (Sept 25, 1995): 53-58.

Author Darleen Sadoski, GTE
sadoski.darleen@mail.ndhm.gtegsc.com

External Reviewer(s) Frank Rogers, GTE

Last Modified 10 Jan 97

Maintainability Index Technique for Measuring Program Maintainability

COMPLETE

Purpose and Origin

Quantitative measurement of an operational system's *maintainability* is desirable both as an instantaneous measure and as a predictor of maintainability over time. Efforts to measure and track maintainability are intended to help reduce or reverse a system's tendency toward "code entropy" or degraded integrity, and to indicate when it becomes cheaper and/or less risky to rewrite the code than to change it. *Software Maintainability Metrics Models in Practice* is the latest report from an ongoing, multi-year joint effort (involving the Software Engineering Test Laboratory of the University of Idaho, the Idaho National Engineering Laboratory, Hewlett-Packard, and other companies) to quantify maintainability via a Maintainability Index (MI) [Welker 95]. Measurement and use of the MI is a process technology, facilitated by simple tools, that in implementation becomes part of the overall development or maintenance process. These efforts also indicate that MI measurement applied during software development can help reduce lifecycle costs. The developer can track and control the MI of code as it is developed, and then supply the measurement as part of code delivery to aid in the transition to maintenance.

Other studies to define code maintainability in various environments have been done [Peercy 81, Bennett 93], but the set of reports leading to the MI measurement technique offered by Welker [Welker 95] describes a method that appears to be very applicable to today's Department of Defense (DoD) systems.

Technical Detail

The literature of at least the last ten years shows that there have been several efforts to characterize and quantify software maintainability; Maintenance of Operational Systems— an Overview (pg. 237) provides a broad overview of software maintenance issues. In this specific technology, a program's maintainability is calculated using a combination of widely-used and commonly-available measures to form a Maintainability Index (MI). The basic MI of a set of programs is a polynomial of the following form (all are based on average-per-code-module measurement):

171 - 5.2 * ln(aveV) - 0.23 * aveV(g') - 16.2 * ln (aveLOC) - 50 * sin (sqrt(2.4 * perCM))

The coefficients are derived from actual usage (see Usage Considerations, pg. 232). The terms are defined as follows:

aveV = average Halstead Volume V per module (see Halstead Complexity Measures, pg. 209)

aveV(g') = average extended cyclomatic complexity per module (see Cyclomatic Complexity, pg. 145)

aveLOC = the average count of lines of code (LOC) per module; and, optionally

perCM = average percent of lines of comments per module

Oman develops the MI equation forms and their rationale [Oman 92a]; the Oman study indicates that the above metrics are good and sufficient predictors of maintainability. Oman builds further on this work using a modification of the MI and describing how it was calibrated for a specific large suite of industrial-use operational code [Oman 94]. Oman describes a prototype tool that was developed specifically to support capture and use of maintainability measures for Pascal and C [Oman 91]. The aggregate strength of this work and the underlying simplicity of the concept make the MI technique potentially very useful for operational Department of Defense (DoD) systems.

Usage Considerations

Calibration of the equations. The coefficients shown in the equation are the result of calibration using data from numerous software systems being maintained by Hewlett-Packard. Detailed descriptions of how the MI equation was calibrated and used appear in Coleman, Pearse, and Welker [Coleman 94, Coleman, 95, Pearse 95, Welker 95]. The authors claim that follow-on efforts show that this form of the MI equation generally fits other industrial-sized software systems [Oman 94 and Welker 95], and the breadth of the work tends to support this claim. It is advisable to test the coefficients for proper fit with each major system to which the MI is applied.

Effects from comments in code. The user must analyze comment content and quality in the specific system to decide whether the comment term perCM is useful.

Ways of using MI

1. The system can be checked periodically for maintainability, which is also a way of calibrating the equations.
2. It can be integrated into a development effort to screen code quality as it is being built and modified; this could yield potentially significant life cycle cost savings.
3. It can be used to drive maintenance activities by evaluating modules either selectively or globally to find high-risk code.
4. MI can be used to compare or evaluate systems: Comparing the MIs of a known-quality system and a third-party system can provide key information in a make-or-buy decision.

Example of usage. Welker relates how a module containing a routine with some "very ugly" code was assessed as unmaintainable, when expressed in terms of the MI (note that just quantifying the problem is a step forward) [Welker 95]. The module was first redesigned, and then functionally enhanced. The measured results are shown in the table below:

Measure	Initial Code		Restructured Code		After Enhancement	
Code Unit	Routine	Module	Routine	Module	Routine	Module
MI (larger MI = more maintainable)	6.47	33.55	39.93	70.13	37.62	69.60
Halstead Effort[a]	2,216,499	2,233,072	182,216	480,261	201,429	499,474
Extended Cyclomatic Complexity[b]	45	49	18	64	21	67
Lines of Code	622	663	196	732	212	748

a. Halstead Effort, rather that Halstead Volume, was used in this case study. See pg. 209 for more information on both these measures. Generally, the lower a program's measure of effort, the simpler a change to the program will be (because Halstead measures are weighted toward measuring computational complexity, not all programs will behave this way).

b. Note that a low Cyclomatic Complexity (see pg. 145) is generally indicative of a lower risk, hence more maintainable, program. In this case, restructuring increased the module complexity slightly (from 49 to 64), but reduced the "ugly" routine's complexity significantly. In both, the subsequent enhancement drove the complexity slightly higher.

If the enhancement had been made without first doing the restructuring, these figures indicate the change would have been much more risky.

Coleman, Pearse, and Welker provide detailed descriptions of how MI was calibrated and used at Hewlett-Packard [Coleman 94, Coleman 95, Pearse 95, Welker 95].

Maturity

Oman tested the MI approach by using production operational code containing around 50 KLOC to determine the metric parameters, and by checking the results against subjective data gathered using the 1989 AFOTEC maintainability evaluation questionnaire [AFOTEC 89, Oman 94]. Other production code of about half that size was used to check the results, with apparent consistency.

Welker applied the results to analyses of a US Air Force (USAF) system, the Improved Many-On-Many (IMOM) electronic combat modeling system. The original IMOM (in FORTRAN) was translated to C and the C

version was later reengineered into Ada. The maintainability of both newer versions was measured over time using the MI approach [Welker 95]. Results were as follows:

- The *reengineered* version's MI was more than twice as high as the original code (larger MI = more maintainable), and declined only slightly over time (note that the original code was not measured over time for maintainability, so change in its MI could not be measured).
- The *translated* baseline's MI was *not* significantly different from the original. This is of special interest to those considering translation, because one of the primary objectives of translation is to reduce future maintenance costs. There was also evidence that the MI of translated code deteriorates more quickly than reengineered code.

Costs and Limitations

Calculating the MI is generally simple and straightforward, given that several commercially-available programming environments contain utilities to count code lines, comment lines, and even cyclomatic complexity (see pg. 145). Other than the tool described in Oman [Oman 91], tools to calculate Halstead measures (see pg. 209) are less common because the measure is not used as widely. However, once conventions for the counting have been established, it is generally not difficult to write language-specific code scanners to count the Halstead components (operators and operands) and calculate the E and V measures. In relating that removal of unused code in a single module did not affect the MI, Pearse highlights the fact that MI is a system measurement; its parameters are average values [Pearse 95]. However, measuring the MI of individual modules is useful because changes in either structural or computational complexity are reflected in a module's MI. A product/process measurement program not already gathering the metrics used in MI could find them useful additions. Those metrics already being gathered may be useful in constructing a custom MI for the system. However, it would be advisable to consult the references for their findings on the effectiveness of metrics, other than Halstead E and V and cyclomatic complexity, in determining maintainability.

Dependencies

The MI method depends on the use of cyclomatic complexity (see pg. 145) and Halstead complexity measures (see pg. 209). To realize the full benefit of MI, the maintenance environment must allow the rewriting of a module when it becomes measurably unmaintainable. The point of measuring the MI is to identify risk; when unacceptably risky code is identified, it should be rewritten.

Alternatives

The process described by Sittenauer is designed to assist in deciding whether or not to reengineer a system [Sittenauer 92]. There are also many research and analytic efforts that deal with maintainability as a

function of program structure, design, and content, but none was found that was as clearly appropriate as MI to current DoD systems in the life-cycle phases described in Maintenance of Operational Systems— an Overview, pg. 237.

Complementary Technologies

The test in Sittenauer is meant to verify generally the condition of a system, and would be useful as a periodic check of a software system and to compare to the MI [Sittenauer 92].

Index Categories

Name of technology	Maintainability Index Technique for Measuring Program Maintainability
Application category	Debugger (AP.1.4.2.4), Test (AP.1.4.3), Unit Testing (AP.1.4.3.4), Component Testing (AP.1.4.3.5), Reapply Software Life Cycle (AP.1.9.3), Reengineering (AP.1.9.5)
Quality measures category	Maintainability (QM.3.1), Testability (QM.1.4.1), Understandability (QM.3.2)
Computing reviews category	Software Engineering Distribution and Maintenance (D.2.7), Software Engineering Metrics (D.2.8), Complexity Classes (F.1.3), Tradeoffs Among Complexity Measures (F.2.3)

References and Information Sources

[AFOTEC 89] *Software Maintainability Evaluation Guide* 800-2, Volume 3. Kirtland AFB, NM: HQ Air Force Operational Test and Evaluation Center (AFOTEC), 1989.

[Ash 94] Ash, Dan, et al. "Using Software Maintainability Models to Track Code Health," 154-160. *Proceedings of the International Conference on Software Maintenance.* Victoria, BC, Canada, September 19-23, 1994. Los Alamitos, CA: IEEE Computer Society Press, 1994.

[Bennett 93] Bennett, Brad & Satterthwaite, Paul. "A Maintainability Measure of Embedded Software," 560-565. *Proceedings of the IEEE 1993 National Aerospace and Electronics Conference.* Dayton, OH, May 24-28, 1993. New York, NY: IEEE, 1993.

[Coleman 94] Coleman, Don, et al. "Using Metrics to Evaluate Software System Maintainability." *Computer 27,* 8 (August 1994): 44-49.

[Coleman 95] Coleman, Don; Lowther, Bruce; & Oman, Paul. "The Application of Software Maintainability Models in Industrial Software Systems." *Journal of Systems Software 29,* 1 (April 1995): 3-16.

[Oman 91] Oman, P. *HP-MAS: A Tool for Software Maintainability* (91-08-TR). Moscow, ID: Software Engineering Test Lab, University of Idaho, 1992.

[Oman 92a] Oman, P. & Hagemeister, J. *Construction and Validation of Polynomials for Predicting Software Maintainability* (92-01TR). Moscow, ID: Software Engineering Test Lab, University of Idaho, 1992.

[Oman 92b] Oman, P. & Hagemeister, J. "Metrics for Assessing a Software System's Maintainability," 337-344. *Conference on Software Maintenance 1992.* Orlando, FL, November 9-12, 1992. Los Alamitos, CA: IEEE Computer Society Press, 1992.

[Oman 94] Oman, P. & Hagemeister, J. "Constructing and Testing of Polynomials Predicting Software Maintainability." *Journal of Systems and Software 24,* 3 (March 1994): 251-266.

[Pearse 95] Pearse, Troy & Oman, Paul. "Maintainability Measurements on Industrial Source Code Maintenance Activities," 295-303. *Proceedings. of the International Conference on Software Maintenance.* Opio, France, October 17-20, 1995. Los Alamitos, CA: IEEE Computer Society Press, 1995.

[Peercy 81] Peercy, David E. "A Software Maintainability Evaluation Methodology." *Transactions on Software Engineering 7,* 7 (July 1981): 343-351.

[Sittenauer 92] Sittenauer, Chris & Olsem, Mike. "Time to Reengineer?" *Crosstalk, Journal of Defense Software Engineering* 32 (March 1992): 7-10.

[Welker 95] Welker, Kurt D. & Oman, Paul W. "Software Maintainability Metrics Models in Practice," *Crosstalk, Journal of Defense Software Engineering 8,* 11 (November/December 1995): 19-23.

[Zhuo 93] Zhuo, Fang, et al. "Constructing and Testing Software Maintainability Assessment Models," 61-70. *Proceedings of the First International Software Metrics Symposium.* Baltimore, MD, May 21-22, 1993. Los Alamitos, CA: IEEE Computer Society Press, 1993.

Author Edmond VanDoren, Kaman Sciences
bvandoren-cos3@kaman.com

External Reviewer(s) Paul W. Oman, Ph.D., Computer Science Department, University of Idaho, Moscow, ID

Kurt Welker, Lockheed Martin, Idaho Falls, ID

Last Modified 10 Jan 97

Maintenance of Operational Systems— an Overview COMPLETE

Note

This description provides background information for technologies for optimizing maintenance environments. We recommend Cyclomatic Complexity, pg. 145; Halstead Complexity Measures, pg. 209; Maintainability Index Technique for Measuring Program Maintainability, pg. 231; and Function Point Analysis, pg. 195, as concurrent reading, as they contain information about specific technologies.

Purpose and Origin

Technologies specific to the maintenance of software evolved (and are still evolving) out of development-oriented technologies. As large systems have proliferated and aged, the special needs of the operational environment have begun to emerge. Maintenance is defined here as the modification of a software product after delivery to correct faults, improve performance or other attributes, or to adapt the product to a changed environment [IEEE 83]. Historically, the software lifecycle has usually focused on development. However, so much of a system's cost is incurred during its operational lifetime that maintenance issues have become more important and, arguably, this should be reflected in development practices. Systems are required to last longer than originally planned; inevitably, the percentage of costs going to maintenance has been steadily climbing. Hewlett-Packard estimates that 60% to 80% of its R&D personnel are involved in maintaining existing software, and that 40% to 60% of production costs were directly related to maintenance [Coleman 94]. There was a rule of thumb that eighty percent of a Department of Defense (DoD) system's cost is in maintenance; older Cheyenne Mountain Complex systems may have surpassed ninety percent. Yet software development practices still do not put much emphasis on making the product highly maintainable.

Cost and risk of maintenance of older systems are further exacerbated by a shortage of suitable maintenance skills; analysts and programmers are not trained to deal with these systems. Industry wide, it is claimed that 75%-80% of all operational software was written without the discipline of structured programming [Coleman 95]. Only a minuscule fraction of current operational systems were built using the object-oriented techniques taught today.

The purpose of this description is to provide a framework or a contextual reference for some of the maintenance and reengineering technologies described in this document.

Technical Detail

The operational system lifecycle. The operational environment has its own lifecycle that, while connected to the development lifecycle, has specific and unique characteristics and needs. As shown in Figure 15, a system's total lifecycle is defined as having four major phases:

- the development or pre-delivery phase
- the early operational phase
- the mature operational phase
- the evolution/replacement phase

Each of the phases has typical characteristics and problems. The *operational* phases are most of the lifecycle and cost. The narrative following describes each phase, and identifies specific technologies in (or planned for) this document that can be applied to correct or improve the situation. In almost every case, taking the proper action in a given phase can eliminate, or greatly reduce, problems in a later phase— at much less cost.

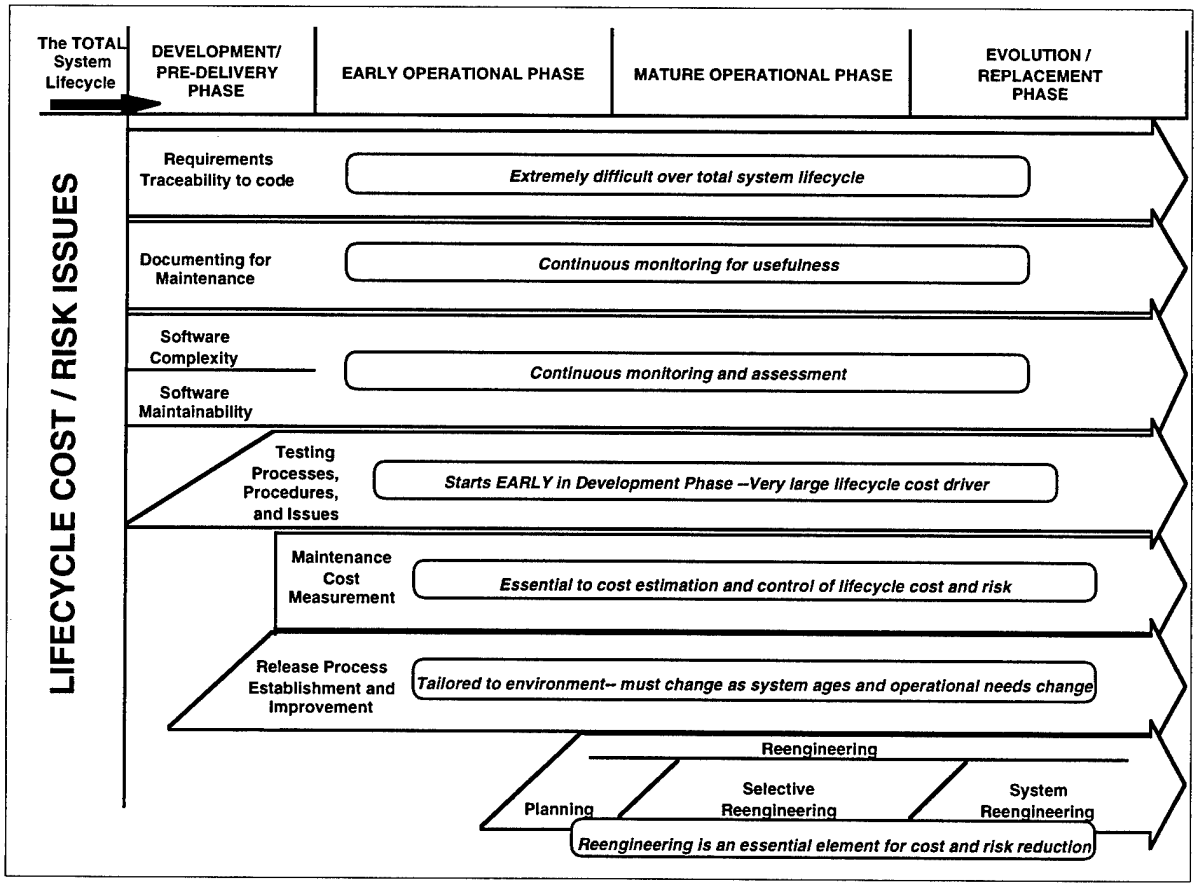


Figure 15: Total System Life Cycle

Terminology. To set a baseline for the descriptions of these phases, the following definitions are used:

Reengineering: rebuilding a piece of software to suit some new purpose (to work on another platform, to switch to another language, to make it more maintainable, etc.); often preceded by reverse engineering. Examination and alteration of a subject system to reconstitute it in a new form. Any activity that improves one's understanding of software, or prepares or improves the software itself for increased maintainability, reusability, or evolvability.

Restructuring: transformation of a program from one representation to another at the same relative abstraction level, usually to simplify or clarify it in some way (e.g., remove GOTOs, increase modularity), while preserving external behavior.

Reverse engineering: the process of analyzing a system's code, documentation, and behavior to identify its current components and their dependencies to extract and create system abstractions and design information. The subject system is not altered; however, additional knowledge about the system is produced. Redocumenting and design recovery are techniques associated with reverse engineering.

Software complexity: some measure of the mental effort required to understand a piece of software.

Software maintainability: some measure of the ease and/or risk of making a change to a piece of software. The measured complexity of the software is often used in quantifying maintainability.

Translation: conversion of a program from one language to another, often as a companion action to restructuring the program.

Phase 1: The development or pre-delivery phase, when the system is not yet operational. Most of the effort in this phase goes into making Version One of the system function. But if total lifecycle costs are to be minimized, planning and preparation for maintenance during the development phase are essential. Most currently operational systems did not receive this attention during development. Several areas should be addressed:

- *Requirements traceability to code.* Requirements are the foundation of a system, and one of the most common faults of an operational system is that the relationship between its requirements and its code cannot be determined. Recovering this information for a system after it goes operational is a costly and time-consuming task. See

Requirements Tracing (pg. 327), Feature-Based Design Rationale Capture Method for Requirements Tracing (pg. 181), and Argument-Based Design Rationale Capture Methods for Requirements Tracing (pg. 91) for assistance in creating initial mapping from requirements to code.

- *Documentation and its usefulness in maintenance.* The ostensible purpose of documentation is to aid in understanding what the system does, and (for the maintenance programmer) how the system does it. There is at least anecdotal evidence that
  - Classical specification-type documentation is not a good primary source of information for the maintenance programmer looking for a problem's origin, especially since the documentation is frequently inconsistent with the code.
  - The most useful maintenance information is derived directly and automatically from the code; examples include structure charts, program flow diagrams, and cross-reference lists. This suggests that tools that create and maintain these documentation forms should be used during development of the code, and delivered with it.
- *The complexity of the software.* If the software is too complex to understand when it is first developed, it will only become more complex and brittle as it is changed. Measuring complexity during code development is useful for checking code condition, helps in quantifying testing costs, and aids in forecasting future maintenance costs (see Cyclomatic Complexity (pg. 145), Halstead Complexity Measures (pg. 209), and Maintainability Index Technique for Measuring Program Maintainability (pg. 231)).
- *The maintainability of the software.* This is perhaps the key issue for the maintainer. The ability to measure a system's maintainability directly affects the ability to predict future costs and risks. Maintainability Index Technique for Measuring Program Maintainability (pg. 231) describes a practical approach to such a measurement, applicable throughout the lifecycle.

Phase 2: The early operational phase, when the delivered system is being maintained and changed to meet new needs and fix problems. Typically the tools and techniques used for maintenance are those that were used to develop the system. In this phase, the following issues are critical:

- *Complexity and maintainability* must be measured and controlled in this phase if the major problems of Phase 3 are to be avoided. Ideally, this a continuation of the same effort that began in Phase 1, and it depends on the same tools and techniques (see Cyclomatic Complexity (pg. 145), Halstead Complexity Measures (pg. 209), and Maintainability Index Technique for Measuring Program Maintainability (pg. 231)). In a preventative maintenance regime, use of these types of measures will help establish guidelines about how

much complexity and/or deterioration of maintainability is tolerable. If a critical module becomes too complex under the guidelines, it should be considered for rework before it becomes a problem. Early detection of problems, such as risk due to increasing complexity of a module, is far cheaper than waiting until a serious problem arises.

- *A formal release-based maintenance process* that suits the environment must be established. This process should always be subject to inspection, and should be revised when it does not meet the need.
- *The gathering of cost data* must be part of the maintenance process if lifecycle costs are to be understood and controlled. The cost of each change (e.g., person-hours, computer-hours) should be known down to a suitable granularity such as phase within the release (e.g., design, code and unit test, integration testing). Without this detailed cost information, it is very hard to estimate future workload or the cost of a proposed change.

Phase 3: Mature operational phase, in which the system still meets the users' primary needs but is showing signs of age. For example

- The incidence of bugs caused by changes or "day-one errors" (problems that existed at initial code delivery) is rising, and the documentation, especially higher-level specification material, is not trustworthy. Most analyses of changes to the software must be done by investigating the code itself.
- Code "entropy" and complexity are increasing and, even by subjective measures, its maintainability is decreasing.
- New requirements increasingly uncover limitations that were designed into the system.
- Because of employee turnover, the programming staff may no longer be intimately familiar with the code, which increases both the cost of a change and the code's entropy.
- A change may have a ripple effect: Because the true nature of the code is not well known, coupling across modules has increased and made it more likely that a change in one area will affect another area. It may be appropriate to restructure or reengineer selected parts of the system to lessen this problem.
- Testing has become more time-consuming and/or risky because as code complexity increases, test path coverage also increases. It may be appropriate to consider more sophisticated test approaches (see Preventive Maintenance, pg. 242)
- The platform is obsolete: The hardware is not supported by the manufacturer and parts are not readily available; the COTS software is not supported through new releases (or the new releases will not

work with the application, and it is too risky to make the application changes needed to align with the COTS software).

At this point, the code has not been rewritten en masse or reverse engineered to recover design, but the risk and cost of evolution by modification of the system have increased significantly. The system has become brittle with age. It may be appropriate to assess the system's condition. Sittenauer describes a quick methodology for gauging the need for reengineering, and the entire approach for measuring maintainability (see pg. 231) allows continuous or spot assessment of the system's maintainability [Sittenauer 92].

Phase 4: Evolution/Replacement Phase, in which the system is approaching or has reached insupportability. The software is no longer maintainable. It has become so "entropic" or brittle that the cost and/or risk of significant change is too high, and/or the host hardware/software environment is obsolete. Even if none of these is true, the cost of implementing a new requirement is not tolerable because it takes too long under the maintenance environment. It is time to consider reengineering (see Cleanroom Software Engineering, pg. 95 and Graphical User Interface Builders, pg. 205).

Usage Considerations

Software maintainability factors. The characteristics influencing or determining a system's maintainability have been extensively studied, enumerated, and organized. One thorough study is described in Oman; such characteristics were analyzed and a simplified maintainability taxonomy was constructed [Oman 91]. Maintainability Index Technique for Measuring Program Maintainability (pg. 231) describes an approach to measuring and controlling code maintainability that was founded on several years of work and analysis and includes analysis of commercial software maintenance. References to other maintainability research results also appear in that technology description.

Preventive maintenance approaches. The approaches listed below are a few of the ways current technology can help to enhance system maintainability.

- *Complexity analysis.* Before attempting to reach a destination, it is essential to know where you are. For a software system, a good first step is measuring the complexity of the component modules (see Cyclomatic Complexity (pg. 145) and Halstead Complexity Measures (pg. 209)). Maintainability Index Technique for Measuring Program Maintainability (pg. 231) describes a method of assessing maintainability of code using those complexity measures. Test path coverage can also be determined from complexity measures, which

can help in optimizing system testing (see Test generation and optimization, 244).

- *Functionality analysis.* Function Point Analysis (pg. 195) describes the uses and limitations of function point analysis (also known as functional size measurement) in measuring software. By measuring a program's functionality, one can arrive at some estimate of its value in a system, which is of use when making decisions about rewriting the program or reengineering the system. Measures of functionality can also guide decisions about where to put testing effort (see Test generation and optimization, 244).
- *Reverse engineering / design recovery.* Over time, a system's code diverges from the documentation; this is a well-known tendency of operational systems. Another phenomenon that is frequently underestimated or ignored is that (regardless of the divergence effect) the information required to make a given change is often found only in the code. Several approaches are possible here. Various tools offer the ability to construct program flow diagrams (PFDs) from code. More sophisticated techniques, often classified as program understanding, are emerging. These technologies are implemented as tools that act as agents for the human analyst to assist in gathering information about a program's function at higher levels of abstraction than a program flow diagram (e.g., retask a satellite).
- *Piecewise reengineering.* If the system's known lifetime is sufficiently short, and if the evolutionary changes needed are sufficiently bounded, the system may benefit from a piecewise reengineering approach:
  - Brittle, high-risk modules that are likely to need changes are identified and reengineered to make them more maintainable. Techniques such as wrappers, an emerging technology, are expected to aid here.
  - For the sake of prudence, other risky modules are "locked," so that a prospective change to them can be made only after thoroughly assessing the risks involved.
  - For database systems, it may be possible to retrofit a modern relational or object-oriented database to the system; Common Object Request Broker Architecture (pg. 107) and Graphic Tools for Legacy Database Migration (pg. 201) describe technologies of possible use here.

  Piecewise reengineering can generally be done at a lower cost than complete reengineering of the system. If it is the right choice, it delays the inevitable obsolescence. The downsides of piecewise reengineering include the following:
  - Platform obsolescence is not reversed. Risks arising from the platform's software are unchanged; if the original database or operating system has risks, the application using them will also.

- Unforeseen requirements changes still carry high risk if they affect the old parts of the system.
- Performance may suffer because of the interface structures added to splice reengineered functions to old ones.

- *Translation/restructuring/modularizing.* Translation and/or restructuring of code are often of interest when migrating software to a new platform. Frequently the new environment will not support the old language or dialect. Restructuring/modularizing, or rebuilding the code to reduce complexity, can be done simply to improve the code's maintainability, but code to be translated is often restructured first so that the result will be less complex and more easily understood. There are several commercial tools that do one or more of these operations, and energetic research to achieve more automated approaches is being done. Welker cites evidence that translation does little or nothing to enhance maintainability [Welker 95]. Most often, it simply continues the existing problem in a different syntactical form; the mechanical forms output by translators decrease understandability, which is a key component of maintainability. None of these technologies is a cure-all, and none of them should be applied without first assessing the quality of the output and the amount of programmer resources required.
- *Test generation and optimization.* Mission criticality of many DoD systems drives the maintenance activity to test very thoroughly. Boehm reported integration testing activities consuming only 16-34% of project totals [Boehm 81], but other evidence is available to show that commercial systems testing activity can take half of a development effort's resources [Alberts 76, DeMillo 87, Myers 79]. Recent composite post-release reviews of operational Cheyenne Mountain Complex system releases show that testing consumed 60-70% of the total release effort.[1] Any technology that can improve testing efficiency will have high leverage on the system's life-cycle costs. Technologies that can possibly help include: automatic test case generation; generation of test and analysis tools; redundant test case elimination; test data generation by chaining; techniques for software regression testing; and techniques for statistical test plan generation and coverage analysis.

1. Source: Kaman Sciences Corp. *Minutes of the 96-1 Composite Post-Release Review (CPRR), Combined CSS/CSSR and ATAMS Post-Release Review and Software Engineering Post-Release Review* KSWENG Memo # 96-03, 26 July, 1996.

Index Categories

Name of technology	Maintenance of Operational Systems— an Overview
Application category	Requirements Tracing (AP.1.2.3), Cost Estimation (AP.1.3.7), Test (AP.1.4.3), System Testing (AP.1.5.3.1), Regression Testing (AP.1.5.3.4), Reapply Software Lifecycle (AP.1.9.3), Reverse Engineering (AP.1.9.4), Reengineering (AP.1.9.5)
Quality measures category	Maintainability (QM.3.1)
Computing reviews category	Software Engineering Distribution and Maintenance (D.2.7), Software Engineering Metrics (D.2.8), Software Engineering Management (D.2.9)

References and Information Sources

[Alberts 76] Alberts, D. "The Economics of Software Quality Assurance." *National Computer Conference*. New York, NY, June 7-10, 1976. Montvale, NJ: American Federation of Information Processing Societies Press, 1976.

[Boehm 81] Boehm, Barry W. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall, 1981.

[Coleman 94] Coleman, Don, et al. "Using Metrics to Evaluate Software System Maintainability." *Computer 27,* 8 (August 1994): 44-49.

[Coleman 95] Coleman, Don; Lowther, Bruce; & Oman, Paul. "The Application of Software Maintainability Models in Industrial Software Systems." *Journal of Systems Software 29,* 1 (April 1995): 3-16.

[DeMillo 87] DeMillo, R., et al. *Software Testing and Evaluation*. Menlo Park, CA: Benjamin/Cummings, 1987.

[IEEE 83] *IEEE Standard Glossary of Software Engineering Terminology*. New York, NY: Institute of Electrical and Electronic Engineers, 1983.

[Myers 79] Myers, G. *The Art of Software Testing*. New York, NY: John Wiley and Sons, 1979.

[Oman 91] Oman, P.; Hagermeister, J.; & Ash, D. *A Definition and Taxonomy for Software Maintainability* (91-08-TR). Moscow, ID: Software Engineering Test Laboratory, University of Idaho, 1991.

[Sittenauer 92] Sittenauer, Chris & Olsem, Mike. "Time to Reengineer?" *Crosstalk, Journal of Defense Software Engineering* 32 (March 1992): 7-10.

[Welker 95] Welker, Kurt D. & Oman, Paul W. "Software Maintainability Metrics Models in Practice." *Crosstalk, Journal of Defense Software Engineering 8*, 11 (November/December 1995): 19-23.

Author Edmond VanDoren, Kaman Sciences
bvandoren-cos3@kaman.com

External Reviewer(s) Brian Gallagher, SEI
Ed Morris, SEI
Dennis Smith, SEI

Last Modified 10 Jan 97

Message-Oriented Middleware Technology ADVANCED

Note *We recommend Middleware, pg. 251, as prerequisite reading for this technology description.*

Purpose and Origin Message-oriented middleware (MOM) is a client/server (see pg. 101) infrastructure that increases the *interoperability*, *portability*, and *flexibility* of an application by allowing the application to be distributed over multiple heterogeneous platforms. It reduces the *complexity* of developing applications that span multiple operating systems and network protocols by insulating the application developer from the details of the various operating system and network interfaces— application programming interfaces (APIs) (see pg. 79) that extend across diverse platforms and networks are typically provided by the MOM [Rao 95].

Technical Detail Message-oriented middleware, as shown in Figure 16 [Steinke 95], is software that resides in both portions of a client/server architecture and typically supports asynchronous calls between the client and server applications. Message queues provide temporary storage when the destination program is busy or not connected. MOM reduces the involvement of application developers with the *complexity* of the master-slave nature of the client/server mechanism.

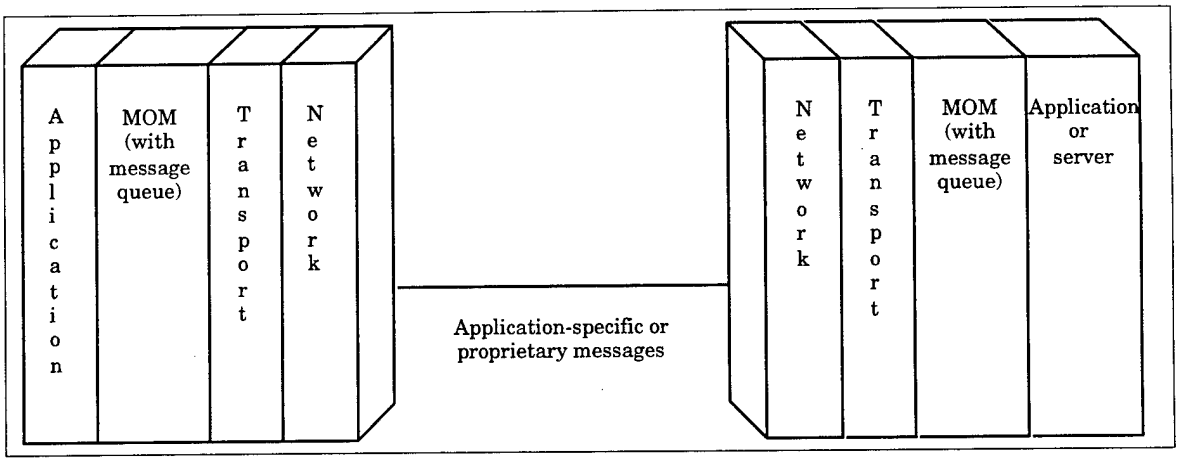


Figure 16: Message-Oriented Middleware

MOM increases the flexibility of an architecture by enabling applications to exchange messages with other programs without having to know what platform or processor the other application resides on within the network. The aforementioned messages can contain formatted data, requests for action, or both. Nominally, MOM systems provide a message queue between interoperating processes, so if the destination process is busy, the

message is held in a temporary storage location until it can be processed. MOM is typically asynchronous and peer-to-peer, but most implementations support synchronous message passing as well.

Usage Considerations

MOM is most appropriate for event-driven applications. When an event occurs, the client application hands off to the messaging middleware application the responsibility of notifying a server that some action needs to be taken. MOM is also well-suited for object-oriented systems because it furnishes a conceptual mechanism for peer-to-peer communications between objects. MOM insulates developers from connectivity concerns—the application developers write to APIs that handle the complexity of the specific interfaces.

Asynchronous and synchronous mechanisms each have strengths and weaknesses that should be considered when designing any specific application.The asynchronous mechanism of MOM, unlike remote procedure call (RPC) (see pg. 323), which uses a a synchronous, blocking mechanism, does not guard against overloading a network. As such, a negative aspect of MOM is that a client process can continue to transfer data to a server that is not keeping pace. Message-oriented middleware's use of message queues, however, tends to be more flexible than RPC-based systems, because most implementations of MOM can default to synchronous and fall back to asynchronous communication if a server becomes unavailable [Steinke 95].

Maturity

Implementations of MOM first became available in the mid-to-late 1980s. Many MOM implementations currently exist that support a variety of protocols and operating systems. Many implementations support multiple protocols and operating systems simultaneously.

Some vendors provide tool sets to help extend existing interprocess communication across a heterogeneous network.

Costs and Limitations

MOM is typically implemented as a proprietary product, which means MOM implementations are nominally incompatible with other MOM implementations. Using a single implementation of a MOM in a system will most likely result in a dependence on the MOM vendor for maintenance support and future enhancements. This could have a highly negative impact on a system's flexibility, maintainability, portability, and interoperability.

The message-oriented middleware software (kernel) must run on every platform of a network. The impact of this varies and depends on the characteristics of the system in which the MOM will be used:

- Not all MOM implementations support all operating systems and protocols. The flexibility to choose a MOM implementation may be dependent on the chosen application platform or network protocols supported, or vice versa.
- Local resources and CPU cycles must be used to support the MOM kernels on each platform. The performance impact of the middleware implementation must be considered; this could possibly require the user to acquire greater local resources and processing power.
- The administrative and maintenance burden would increase significantly for a network manager with a large distributed system, especially in a mostly heterogeneous system.
- A MOM implementation may cost more if multiple kernels are required for a heterogeneous system, especially when a system is maintaining kernels for old platforms and new platforms simultaneously.

Alternatives

Other infrastructure technologies that allow the distribution of processing across multiple processors and platforms are

- object request broker (ORB) (see pg. 291)
- Distributed Computing Environment (DCE) (see pg. 167)
- remote procedure call (RPC) (see pg. 323)
- transaction processing monitor (see pg. 373)
- three tier architectures (see pg. 367)

Complementary Technologies

MOM can be effectively combined with remote procedure call (RPC) technology—RPC can be used for synchronous support by a MOM.

Index Categories

Name of technology	Message-Oriented Middleware Technology
Application category	Client/Server (AP.2.1.2.1), Client/Server Communication (AP.2.2.1)
Quality measures category	Maintainability (QM.3.1), Interoperability (QM.4.1), Portability (QM.4.2)
Computing reviews category	Distributed Systems (C.2.4), Network Architecture and Design (C.2.1)

References and Information Sources

[Rao 95] Rao, B.R. "Making the Most of Middleware." *Data Communications International 24*, 12 (September 1995): 89-96.

[Steinke 95] Steinke, Steve. "Middleware Meets the Network." *LAN Magazine 10,* 13 (December 1995): 56.

Author Cory Vondrak, TRW, Redondo Beach, CA

External Reviewer(s) Ed Morris, SEI

Last Modified 10 Jan 97

Middleware

ADVANCED

Purpose and Origin

Middleware is connectivity software that consists of a set of enabling services that allow multiple processes running on one or more machines to interact across a network. Middleware is essential to migrating mainframe applications to client/server applications and to providing for communication across heterogeneous platforms. This technology has evolved during the 1990s to provide for *interoperability* in support of the move to client/server architectures (see pg. 101). The most widely-publicized middleware initiatives are the Open Software Foundation's Distributed Computing Environment (DCE) (see pg. 167), Object Management Group's Common Object Request Broker Architecture (CORBA) (see pg. 107), and Microsoft's Object Linking and Embedding/Component Object Model (OLE) (see pg. 271) [Eckerson 95].

Technical Detail

As outlined in Figure 17, middleware services are sets of distributed software that exist between the application and the operating system and network services on a system node in the network.

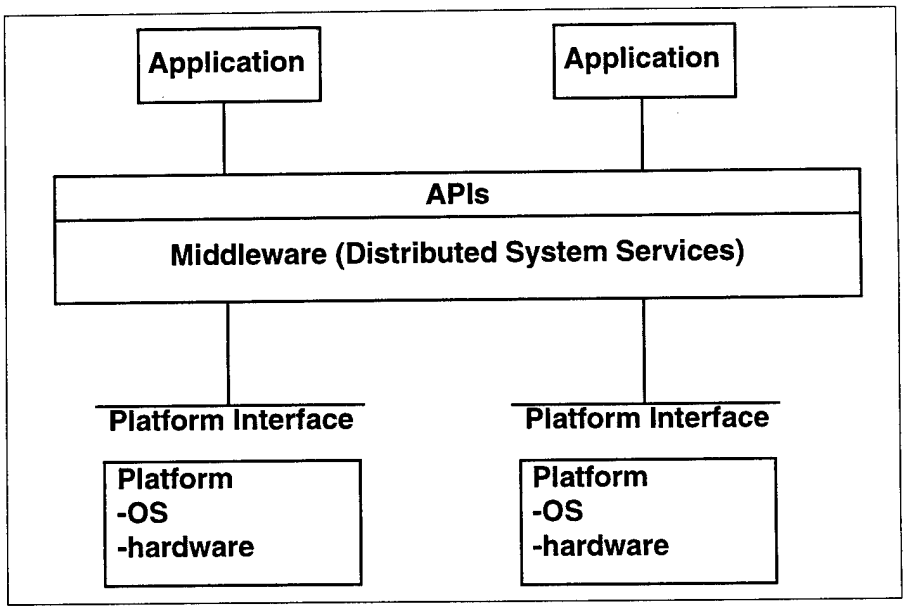


Figure 17: Use of Middleware [Bernstein 96]

Middleware services provide a more functional set of application program interfaces (API) (see pg. 79) than the operating system and network services to allow an application to

- locate transparently across the network, providing interaction with another application or service

- be independent from network services
- be reliable and available
- scale up in capacity without losing function [Schreiber 95]

Middleware can take on the following different forms:

- Transaction processing (TP) monitors (see pg. 373), which provide tools and an environment for developing and deploying distributed applications.
- Remote procedure calls (RPCs) (see pg. 323), which enable the logic of an application to be distributed across the network. Program logic on remote systems can be executed as simply as calling a local routine.
- Message-oriented middleware (MOM) (see pg. 247), which provides program-to-program data exchange, enabling the creation of distributed applications. MOM is analogous to email in the sense it is asynchronous and requires the recipients of messages to interpret their meaning and to take appropriate action.
- Object request brokers (ORBs) (see pg. 291), which enable the objects that comprise an application to be distributed and shared across heterogeneous networks.

Usage Considerations

The main purpose of middleware services is to help solve many application connectivity and interoperability problems. However, middleware services are not a panacea:

- There is a gap between principles and practice. Many popular middleware services use proprietary implementations (making applications dependent on a single vendor's product).
- The sheer number of middleware services is a barrier to using them. To keep their computing environment manageably simple, developers have to select a small number of services that meet their needs for functionality and platform coverage.
- While middleware services raise the level of abstraction of programming distributed applications, they still leave the application developer with hard design choices. For example, the developer must still decide what functionality to put on the client and server sides of a distributed application [Bernstein 96].

The key to overcoming these three problems is to fully understand both the application problem and the value of middleware services that can enable the distributed application. To determine the types of middleware services required, the developer must identify the functions required, which fall into one of three classes:

1. Distributed system services, which include critical communications, program-to-program, and data management services. This type of service includes RPCs, MOMs and ORBs.
2. Application enabling services, which give applications access to distributed services and the underlying network. This type of services includes transaction monitors (see pg. 373) and database services such as Structured Query Language (SQL).
3. Middleware management services, which enable applications and system functions to be continuously monitored to ensure optimum performance of the distributed environment [Schreiber 95].

Maturity

A significant number of middleware services and vendors exist. Middleware applications will continue to grow with the installation of more heterogeneous networks. An example of middleware in use is the Delta Airlines Cargo Handling System, which uses middleware technology to link over 40,000 terminals in 32 countries with UNIX services and IBM mainframes. By 1999, middleware sales are expected to exceed $6 billion [Client 95].

Costs and Limitations

The costs of using middleware technology (i.e., license fees) in system development are entirely dependent on the required operating systems and the types of platforms. Middleware product implementations are unique to the vendor. This results in a dependence on the vendor for maintenance support and future enhancements. This reliance could have a negative effect on a system's flexibility and maintainability. However, when evaluated against the cost of developing a unique middleware solution, the system developer and maintainer may view the potential negative effect as acceptable.

Index Categories

Name of technology	Middleware
Application category	Client/Server (AP.2.1.2.1), Client/Server Communication (AP.2.2.1)
Quality measures category	Interoperability (QM.4.1)
Computing reviews category	Distributed Systems (C.2.4), Network Architecture and Design (C.2.1), Database Management Languages (D.3.2)

References and Information Sources

[Bernstein 96] Bernstein, Philip A. "Middleware: A Model for Distributed Services." *Communications of the ACM 39*, 2 (February 1996): 86-98.

[Client 95] "Middleware Can Mask the Complexity of your Distributed Environment." *Client/Server Economics Letter 2*, 6 (June 1995): 1-5.

[Eckerson 95] Eckerson, Wayne. "Searching for the Middle Ground." *Business Communications Review 25,* 9 (September 1995): 46-50.

[Schreiber 95] Schreiber, Richard. "Middleware Demystified." *Datamation 41,* 6 (April 1, 1995): 41-45.

Author Mike Bray, Lockheed-Martin Ground Systems
michael.w.bray@den.mmc.com

Last Modified 10 Jan 97

Module Interconnection Languages — COMPLETE

Purpose and Origin

As software system size and complexity increase, the task of integrating independently-developed subsystems becomes increasingly difficult. In the 1970s, manual integration was augmented with various levels of automated support, including support from module interconnection languages (MILs). The first MIL, MIL75, was described by DeRemer and Kron [DeRemer 76], who argued with integrators and developers about the differences between programming in the small, for which typical languages are suitable, and programming in the large, for which a MIL is required for knitting modules together [Prieto-Diaz 86]. MILs provide formal grammar constructs for identifying software system modules and for defining the interconnection specifications required to assemble a complete program [Prieto-Diaz 86]. MILs increase the *understandability* of large systems in that they formally describe the structure of a software system; they consolidate design and module assembly in a single language. MILs can also improve the *maintainability* of a large system in that they can be used to prohibit maintainers from accidentally changing the architectural design of a system, and they can be integrated into a larger development environment in which changes in the MIL specification of a system are automatically reflected at the code level and vice versa.

Technical Detail

A MIL identifies the system modules and states how they fit together to implement the system's function; MILs are *not* concerned with what the system does, how the major parts of the system are embedded in the organization, or how the individual modules implement their functions [Prieto-Diaz 86]. A MIL specification of a system constitutes a written description of the system design. A MIL specification can be used to

- Enforce system integrity and inter-modular compatibility.
- Support incremental modification. Modules can be independently compiled and linked; full recompilation of a modified system is not needed.
- Enforce version control. Different versions (implementations) of a module can be identified and used in the construction of a software system. This idea has been generalized to allow different versions of subsystems to be defined in terms of different versions of modules. Thus MILs can be used to describe families of modules and systems [Tichy 79, Cooprider 79].

For example, consider the simplified MIL specification shown in Figure 18 and its associated graphical representation shown in Figure 19. The hypothetical MIL used in Figure 18 contains structures for identifying the modules of interest (in this case the modules are ABC, Z, and YBC); structures for identifying required and provided data; provided functions;

and structures for identifying module and function versions. The module ABC defined in the figure consists of two parts, a function XA and a module YBC; the structure of each of these entities is also defined. Note that function XA has three versions, a Pascal, an Ada, and a FORTRAN version. These three versions would be written and compiled using their respective language development environments. A compilation system for this hypothetical MIL would process the specification given in Figure 18 to check that all required resources (such as x and z) are provided, and to check data type compatibility between required and provided resources. Provided these checks passed, the MIL compilation system, in conjunction with outside (user or environmental) inputs such as version availability and language choices, would select, compile (if necessary), and link the system. Incremental compilation is supported; for example, if the implementations for function XA change, the MIL compilation system will analyze the system structure and recompile and relink only those portions of the overall system affected by that change.

```
Module ABC
   provides a,b,c
   requires x,y
   consists-of function XA, module YBC

   function XA
      must-provide a
      requires x
      has-access-to Module Z
      real x, integer a
      realization
         version Pascal resources file
         (<Pascal_XA>) end Pascal
         version Ada resources file
         (<Ada_XA>) end Ada
         version FORTRAN resources file
         (<FORTRAN_XA>) end FORTRAN
   end XA

   Module YBC
      must-provide b, c
      requires a, y
      real y, integer a, b, c
   end YBC
end ABC
```

Figure 18: MIL Specification of a Simple Module

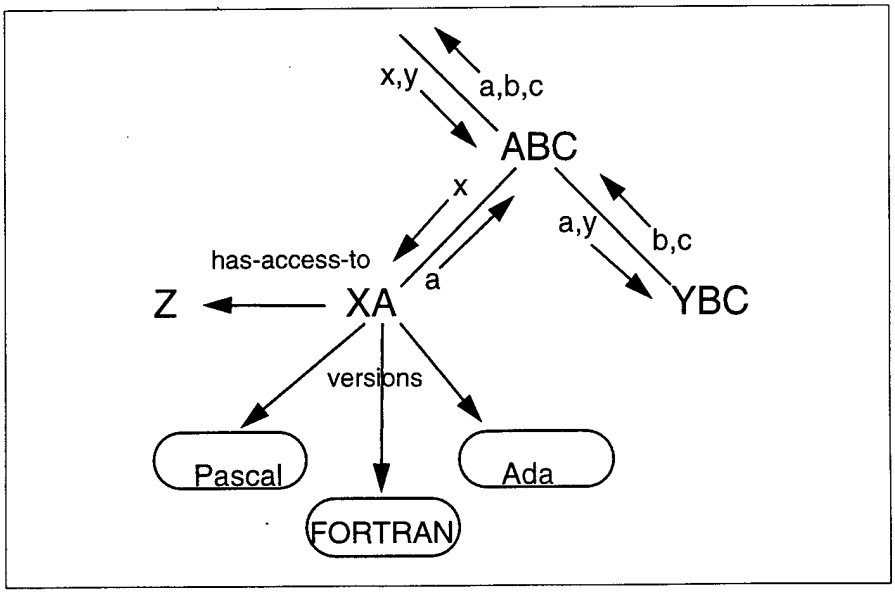


Figure 19: Graphical Representation

MILs do *not* attempt to do the following [Prieto-Diaz 86]:

- *Load compiled images.* This function is left to a separate facility within the development environment.
- Define system function. A MIL defines only the structure, not the function, of a system.
- Provide type specifications. A MIL is concerned with showing or identifying the separate paths of communication between modules. Syntactic checks along these communications paths may be performed by a MIL, but because MILs are independent of the language chosen to implement the modules they reference, such type checking will be limited to simple syntactic— not semantic— compatibility.
- Define embedded link-edit instructions.

Recently, MILs have been extended with notions of communication protocols [Garlan 94] and with constructs for defining semantic properties of system function. These extended MILs are referred to as architecture description languages (ADLs) (see pg. 83).

Usage Considerations

MILs were developed to address the need for automated integration support when programming in the large; they are well-suited for representing the structure of a system or family of systems, and are typically used for project management and support. When adopting the use of MILs, an organization will need to consider the effect on its current system development and maintenance philosophy.

Because the structure of a software system can be explicitly represented in a MIL specification, separate documentation describing software structure may be unnecessary. This implies that if MILs are used to define the structure, then the architectural documentation of a given system will not become outdated.

Although some support is provided for ensuring data type compatibility, MILs typically lack the structures required to define or enforce protocol compatibility between modules, and the structures necessary to enforce semantic compatibility.

Maturity

The MESA system at Xerox PARC was developed during 1975 and has been used extensively within Xerox [Geschke 77, Mitchell 79, Prieto-Diaz 86]. Other MILs have been proposed, defined, and implemented, but most of these appear to have been within a research context. For example, MIL concepts have been used to help design and build software reuse systems such as Goguen's library interconnection language (LIL) that was extended by Tracz for use with parameterized Ada components [Tracz 93]. Zand, et al., describe a system called ROPCO that can be used to "facilitate the selection and integration of reusable modules" [Zand 93].

At the time of publication, however, there are no tools supporting MILs and little research in this area.[1] Recent MIL-based research has shifted focus and now centers around the themes of software reuse and architecture description languages (ADLs). ADLs (see pg. 83) can be viewed as extended MILs in that ADLs augment the structural information of a MIL with information about communication protocols [Garlan 94] and system behavior.

Costs and Limitations

MILs are formal compilable languages. Developers will need training to understand and use a MIL effectively. Training in architectural concepts may also be required.

The lack of a formal semantic for defining module function has at least the following implications:

- *Limited inter-module consistency checking.* MIL-based consistency checking is limited to simple type checking and— if supported— simple protocol checking.
- *Limited consistency checking among module versions.* MILs lack the facilities to ensure that different versions of a module satisfy a

1. Source: Will Tracz in *Re: External Review - MILS*, email to Bob Rosenstein (1996).

common specification, and may potentially lead to inconsistent versions within a family.

- *Limited type checking.* If mixing languages with a system, a developer may need to augment standard MIL tools with more sophisticated type checking utilities. For example, data types may be represented differently in C than in Ada, but the simple type checking found in a typical MIL will not flag unconverted value passing between languages.

Dependencies

Incremental compilers and linkers are required by most MILs.

Alternatives

Alternatives to MILs include documenting the structure of a system externally, such as in an interface control document or a structure chart. Architecture description languages (ADLs) (see pg. 83) can also be used to define the structure of a system, and are believed to be the current direction for this technology area.

Index Categories

Name of technology	Module Interconnection Languages
Application category	Architectural Design (AP.1.3.1), Compiler (AP.1.4.2.3), Plan and Perform Integration (AP.1.4.4)
Quality measures category	Correctness (QM.1.3), Structuredness (QM.3.2.3), Reusability (QM.4.4)
Computing reviews category	Software Engineering Tools and Techniques (D.2.2), Organization and Design (D.4.7), Performance (D.4.8), Systems Programs and Utilities (D.4.9)

References and Information Sources

[Cooprider 79] Cooprider, Lee W. *The Representation of Families of Software Systems* (CMU-CS-79-116). Pittsburgh, PA: Computer Science Department, Carnegie Mellon University, 1979.

[DeRemer 76] DeRemer, F. & Kron, H. "Programming-in-the-Large Versus Programming-in-the-Small." *IEEE Transactions on Software Engineering SE-2,* 2 (June 1976): 321-327.

[Garlan 94] Garlan, David & Allen, Robert. "Formalizing Architectural Connection," 71-80. *Proceedings of the 16th International Conference on Software Engineering.* Sorrento, Italy, May 16-21, 1994. Los Alamitos, CA: IEEE Computer Society Press, 1994.

[Geschke 77] Geschke, C.; Morris, J.; & Satterthwaite, E. "Early Experience with MESA." *Communications of the ACM 20,* 8 (August 1977): 540-553.

[Mitchell 79] Mitchell, J.; Maybury, W.; & Sweet, R. *MESA Language Manual* (CSL-79-3). Palo Alto, CA: Xerox Palo Alto Research Center, April 1979.

[Prieto-Diaz 86] Prieto-Diaz, Ruben & Neighbors, James. "Module Interconnection Languages." *Journal of Systems and Software 6*, 4 (1986): 307-334.

[Tichy 79] Tichy, W. F. "Software Development Control Based on Module Interconnection," 29-41. *Proceedings of the 4th International Conference on Software Engineering.* Munich, Germany, September 17-19, 1979. New York, NY: IEEE Computer Society Press, 1979.

[Tracz 93] Tracz, W. "LILEANNA: a Parameterized Programming Language," 66-78. *Proceedings oFf the Second International Workshop on Software Reuse.* Lucca, Italy, March 24-26, 1993. Los Alamitos, CA: IEEE Computer Society Press, 1993.

[Zand 93] Zand, M., et al. "An Interconnection Language for Reuse at the Template/Module Level." *Journal of Systems and Software 23*, 1 (October 1993): 9-26.

Author Mark Gerken, Rome Laboratory
gerken@ai.rl.af.mil

External Reviewer(s) Will Tracz, Lockheed Martin Federal Systems, Owego, NY

Last Modified 10 Jan 97

Multi-Level Secure Database Management Schemes

ADVANCED

Note

We recommend Computer System Security— an Overview, pg. 129, as prerequisite reading for this technology description.

Purpose and Origin

Conventional database management systems (DBMS) do not recognize different security levels of the data they store and retrieve. They treat all data at the same security level. Multi-level secure (MLS) DBMS schemes provide a means of maintaining a collection of data with mixed security levels. The access mechanisms allow users or programs with different levels of security clearance to store and obtain only the data appropriate to their level.

Technical Detail

As shown in Figure 20, multi-level secure DBMS architecture schemes are categorized into two general types:

- the Trusted Subject architecture
- the Woods Hole architectures

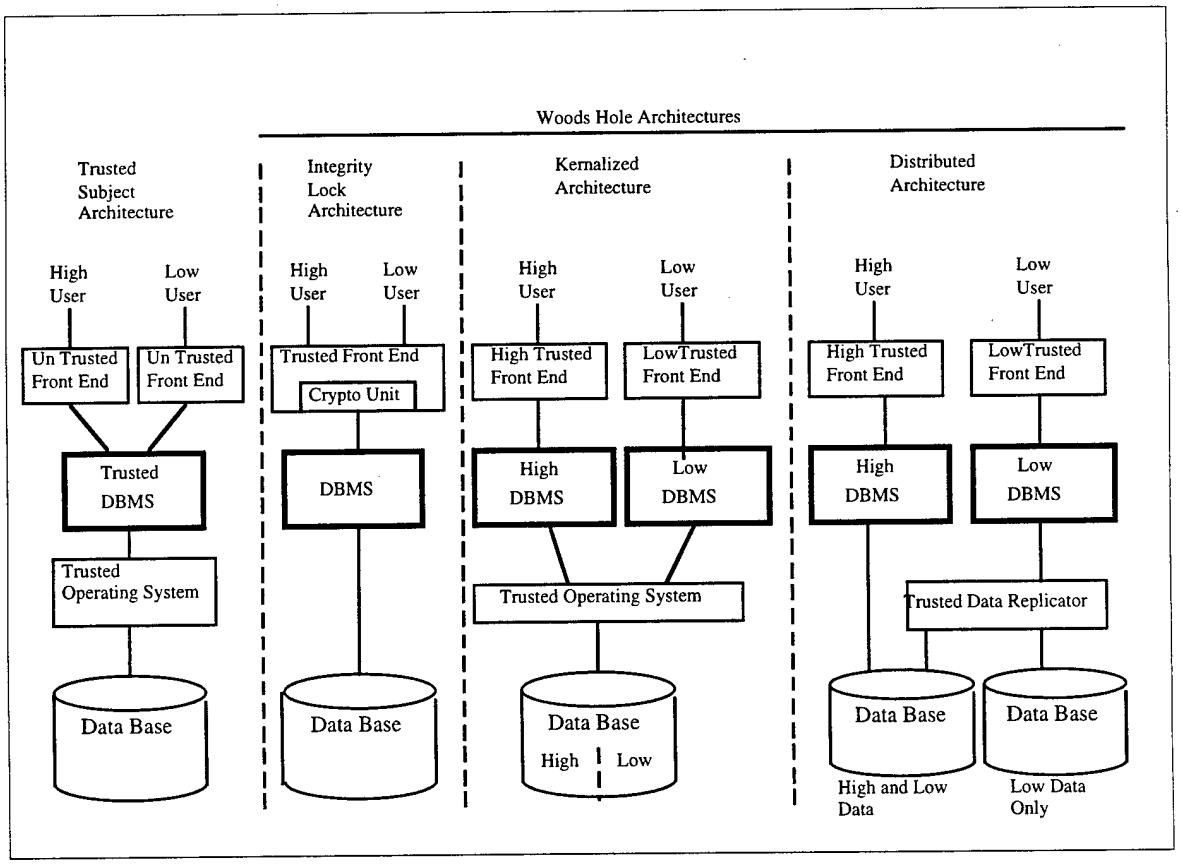


Figure 20: MLS DBMS Schemes

The Woods Hole architectures are named after an Air Force-sponsored study on multi-level data management security that was conducted at Woods Hole, Massachusetts.

The Trusted Subject architecture is a scheme that contains a trusted DBMS and operating system (see pg. 377). The DBMS is custom-developed with all the required security policy (the security rules that must be enforced) developed in the DBMS itself. The DBMS uses the associated trusted operating system to make actual disk data accesses. This is the traditional way of developing MLS DBMS capabilities and can achieve high mandatory assurance for a particular security policy at the sacrifice of some DBMS functionality [Abrams 95]. This scheme results in a special purpose DBMS and operating system that requires a large amount of trusted code to be developed and verified along with the normal DBMS features.Trusted code provides security functionality and has been designed and developed using a rigorous process, tested, and protected from tampering in a manner that ensures the Designated Approving Authority (DAA) that it performs the security functions correctly. The DAA is the security official with the authority to say a system is secure and is permitted to be used. A benefit of the trusted subject architecture is that the DBMS has access to all levels of data at the same time, which minimizes retrieval and update processing. This scheme also can handle a wide range of sensitivity labels and supports complex access control. A sensitivity label identifies the classification level (e.g., confidential, secret) and a set of categories or compartments that apply to the data associated with the label.

The Woods Hole architectures assume that an untrusted (usually commercial-off-the-shelf (COTS)) DBMS is used to access data and that trusted code is developed around that DBMS to provide an overall secure DBMS system. The three different Woods Hole architectures address three different ways to wrap code around the untrusted DBMS.

The Integrity Lock architecture scheme places a trusted front end filter between the users and the DBMS. The filter provides security for the MLS. When data is added to the database, the trusted front end filter adds an encrypted integrity lock to each unit of data added to the database. The lock is viewed by the DBMS as just another element in the unit stored by the DBMS. The encrypted lock is used to assure that the retrieved data has not been tampered with and contains the security label of the data. When data is retrieved, the filter decrypts the lock to determine if the data can be returned to the requester. The filter is designed and trusted to keep users separate and to store and provide data appro-

priate to the user. A benefit of this scheme is that an untrusted COTS DBMS can perform most indexed data storage and retrieval.

The Kernalized architecture scheme uses a trusted operating system and multiple copies of the DBMS; each is associated with a trusted front end. The trusted front end-DBMS pair is associated with a particular security level. Between the DBMS and the database, a portion of the trusted operating system keeps the data separated by security level. Each trusted front end is trusted to supply requests to the proper DBMS. The database is separated by security level. The trusted operating system separates the data when it is added to the database by a DBMS and combines the data when it is retrieved (if allowed by the security rules it enforces for the requesting DBMS). The high DBMS gets data combined from the high and low segments of the database. The low DBMS can only get data from the low segment of the database. A benefit of this scheme is that access control and separation of data at different classification levels is performed by a trusted operating system rather than the DBMS. Data at different security levels is isolated in the database, which allows for higher level assurance. Users interact with a DBMS at the user's single-session level.

The Distributed architecture scheme uses multiple copies of the trusted front end and DBMS, each associated with its own database storage. In this architecture scheme, low data is replicated in the high database. When data is retrieved, the DBMS retrieves it only from its own database. A benefit of this architecture is that data is physically separated into separate hardware databases. Since separate replicated databases are used for each security level, the front end does not need to decompose user query data to different DBMSs.

Castano and Abrams provide thorough discussions of these alternative architecture schemes and their merits [Castano 95, Abrams 95].

Usage Considerations

This technology is most likely to be used when relational databases must be accessed by users with different security clearances. This is typical of Command and Control systems. The different architectures suit different needs. The Trusted Subject architecture is best for applications where the trusted operating system and the hardware used in the architecture already provide an assured, trusted path between applications and the DBMS [Castano 95]. The Integrity Lock architecture provides the ability to label data down to the row (or record) level, the ability to implement a wide range of categories, and is easiest to validate [Castano 95]. The Kernalized architecture scheme is suited to MLS DBMS systems with more simple table structures because it is economical and easier to im-

plement for simple structures [Castano 95]. The Distributed architecture is best suited for DBMSs where physical separation of data by security level is required [Abrams 95].

Maturity

The four different architectures have different maturity characteristics. As of August 1996, an R&D A1[1] system and six commercial[2] DBMSs have been implemented using the Trusted Subject architecture scheme for different assurance levels and security policies. One R&D system and one commercial DBMS have been implemented using the Integrity Lock architecture scheme. One R&D system and one commercial DBMS have been implemented using the Kernalized architecture scheme [Castano 95]. The Distributed architecture scheme has only been used in prototype systems because of the high performance cost of the replicater, although one commercial DBMS claims to have this feature [Abrams 95]. This DBMS however, has not been evaluated by the National Computer Security Center (NCSC) [TPEP 96].

Costs and Limitations

Each of the different MLS architecture schemes has different costs and limitations. The Trusted Subject architecture scheme has a closely linked DBMS and Operating System that must be proven trusted together. This makes it hardest to validate and gives it the highest accreditation cost compared to the other schemes. The Integrity Lock architecture scheme requires that a Crypto Key management system is implemented and supported in operation. The Kernalized architecture requires a DBMS for each security level, which makes it expensive as more than two or three levels are considered. The Distributed architecture requires a different hardware platform for each security level and the data replicater provides a heavy processor and I/O load for high access data.

Dependencies

The MLS architecture schemes have individual dependencies. The Trusted Subject scheme is dependent on trusted schemes for a related DBMS and operating system. The Integrity Lock scheme is dependent on

---

1. An A1 system is one that meets the highest (most stringent) set of requirements in the Department of Defense Trusted Computer Systems Evaluation Criteria (the Orange Book) [DoD 85]. See page 377 for a further description of the classes of trusted operating systems.

2. A commercial DBMS does not imply a general-purpose DBMS. It means that it can be packaged and sold to other people. If a MLS DBMS has been developed to provide specific security functions that customers need, and the customer is willing to be restricted to that set of functions and use the same hardware and support software, then it can be sold as a product. It is then a commercial DBMS. The six commercial DBMSs that have been implemented with the Trusted Subject architecture are all different from each other, as they have been developed with different security policies for different hardware and software environments.

cryptographic technologies to provide the integrity lock. The Kernalized architecture scheme depends on trusted operating system (see pg. 377) technologies. The Distributed architecture scheme is dependent on efficient automatic data replication techniques.

Alternatives

The alternative to these technologies is to use a single-level DBMS and use manual review of retrieved data or have every user cleared for the data in the database. That may not be feasible in a Command and Control system.

Index Categories

Name of technology	Multi-Level Secure Database Management Schemes
Application category	Data Management Security (AP.2.4.2)
Quality measures category	Security (QM.2.1.5)
Computing reviews category	Operating Systems Security & Protection (D.4.6), Security & Protection (K.6.5), Computer-Communications Network Security and Protection (C.2.0)

References and Information Sources

[Abrams 95] Abrams, Marshall D.; Jajodia, Sushil; & Podell, Harold J. *Information Security An Integrated Collection of Essays.* Los Alamitos, CA: IEEE Computer Society Press, 1995.

[Castano 95] Castano, Silvana, et al. *Database Security.* New York, NY: ACM Press, 1995.

[DoD 85] *Department of Defense (DoD) Trusted Computer System Evaluation Criteria (TCSEC)* (DoD 5200.28-STD). Fort George G. Meade, MD: United States Department of Defense, 1985.

[TPEP 96] *Trusted Product Evaluation Program Evaluated Product List* [online]. Available WWW <URL: http://www.radium.ncsc.mil/tpep/index.html> (1996).

Author

Tom Mills, Lockheed Martin
TMILLS@ccs.lmco.com

Last Modified

10 Jan 97

Multi-Level Secure One Way Guard with Random Acknowledgment

DRAFT

Note

We recommend Computer System Security— an Overview, pg. 129, as prerequisite reading for this technology description.

Purpose and Origin

Multi-level secure (MLS) systems are composed of low systems and high systems. Low systems can transmit data to a high system, but high systems cannot transmit data to a low system. That is called *write down* and it is not allowed by multi-level security models, not even to acknowledge (ACK) receipt of data from the low system. This rule exists to prevent a covert timing channel from the high system to the low system. If data integrity and reliable communications are to occur in a system, then messages must be acknowledged. MLS one way guard with random ACK is a form of information flow controls to be imbedded in operational systems that provides a means of acknowledging data without providing a covert path. This technology was first developed (theoretically) in 1993 as an interface between one source and one destination. In 1995 the concept was expanded to address a network of several source low and destination high systems.

Technical Detail

This technology employs a one way guard that buffers a message from a low system and passes it on to the high system. When the high system ACKs the message, the one way guard holds the ACK for a bounded random length of time until passing the ACK to the low system. This destroys any possible covert timing channel as the high system has no control of the timing to the low system. The algorithm to determine the length of time to delay the ACK considers the effect on throughput of delaying multiple sources of data for each destination and the combined throughput to the destination. The algorithm therefore becomes more complex as more sources and destinations are considered. There will be a small negative performance influence on individual messages that could require upgraded interfaces if they are close to capacity. A benefit of this technology is that it allows reliable transmission over an MLS network because messages that are not ACKed are recognized as not received and can then be retransmitted by the sending system.

Usage Considerations

Sending processes using this technology must account for the maximum possible delay in an ACK before retransmitting a message. Increased buffer space must be provided in the one way guard to hold messages until they can be ACKed. The amount of time and amount of buffer space required are a function of the number of sources and destinations involved and the size and rate of messages. Using this technology in a net-

work of mixed security systems provides for no lost messages and no duplication of messages.

Maturity

This technology is new but is an incremental development of one way security guards that have been in use since the 1960s. This technology has been modeled and prototyped but has not been used in an operational system.

Costs and Limitations

Using this technology will require knowledge of security architectures, the recognition of covert timing channels and means to eliminate them, and Designated Approving Authority (DAA) requirements for assurance.[1]

Dependencies

Successful use of this technology in a system requires that an ACK protocol be employed by the nodes that sends another message only after the last transmitted message has been ACKed.

Alternatives

Other approaches to transferring data through a one way guard to enhance reliability involve multiple transmissions of a message without acknowledging receipt or manual accounting of messages and requests for transmission. These alternatives lead to increased traffic over the network because of duplicate messages or increased operator interaction.

Complementary Technologies

A complimentary technology is covert channel analysis in MLS systems.

Index Categories

Name of technology	Multi-Level Secure One Way Guard with Random Acknowledgment
Application category	Information Security (AP.2.4.3)
Quality measures category	Vulnerability (QM.2.1.4.1), Security (QM.2.1.5)
Computing reviews category	Computer-Communications Networks Security and Protection (C.2.0), Security and Protection (K.6.5)

References and Information Sources

[IEEE 95] *Proceedings of the 1995 IEEE Symposium on Security and Privacy.* Oakland, CA, May 8-10, 1995. Los Alamitos, CA: IEEE Computer Society Press, 1995.

Last Modified

10 Jan 97

1. The DAA is the security official with the authority to say a system is secure and is permitted to be used.

Nonrepudiation in Network Communications DRAFT

Note

We recommend Computer System Security— an Overview, pg. 129, as prerequisite reading for this technology description.

Purpose and Origin

The goal of nonrepudiation is to prove that a message has been sent and received. This is extremely important in C4I networks where commands and status must be issued and responded to, in banking networks where financial transactions must be verifiably completed, and in legal networks where signed contracts are transmitted. The Trusted Network Interpretation of the Trusted Computer System Evaluation Criteria (the Red Book) defines the requirement for the military environment. Current technology to accomplish this involves a central authority that verifies and time stamps digital signatures. The technologies for digital signatures have existed since the development of Public Key Cryptography in the late 1970s.

Technical Detail

Three parties are involved in current nonrepudiation schemes: the message sender, the message arbitrator, and the message receiver. The sender creates a message and creates and appends a public key encryption based digital signature to the message. The sender appends identifying data to the message and signs it again. The sender then transmits the message over the network to the arbitrator. The arbitrator verifies the sender's signature and identifying data. The arbitrator then adds a time stamp to the message and signs it. The message is then sent to both the sender and the receiver. The receiver verifies the arbitrator's signature and the sender's signature. The sender verifies the message transmitted by the arbitrator as a copy of the one the sender originally sent. If it does not verify or the sender did not send an original message, the arbitrator is notified immediately. This prevents someone from pretending to be the sender and transmitting a message to the receiver. The arbitrator keeps a record of expired or compromised secret keys to use in the verification process. This whole technology process assures the receiver that the message came from the indicated source and records the time that the message was sent from the sender to the receiver. The sender can not claim to not have sent the message nor that a lost cryptographic key was used. The message sender, arbitrator, and receiver can be implemented in software in different parts of the network.

Usage Considerations

This technology introduces considerable overhead in the processing of messages. Not only are there creation and verification additions at each end of the transmission but the third party arbitrator processing adds additional overhead and delay. The additional overhead should be considered in the design of the system that uses the technology. This

technology may provide the only assured means to identify a source of a message on a network and associate it with a time. The same technology can be used to validate an acknowledgment message.

Maturity

The components of this technology are mature and are used in networks consisting of PCs, workstations, or mainframes.

Costs and Limitations

Using this technology requires knowledge of digital signature algorithms, public key encryption, one-way hashing algorithms and the means of protecting the related keys from inadvertent or malicious compromise.

Dependencies

Successful use of this technology requires the generation and distribution of public keys and the generation and protection of secret keys.

Alternatives

A less secure alternative is to use a time stamp in the senders signature without using a central arbitrator. This is less secure because the sender could claim that someone else sent the message with a stolen or lost key.

Complementary Technologies

Complementary technologies include one-way hashing, digital signatures, and public key cryptography.

Index Categories

Name of technology	Nonrepudiation in Network Communications
Application category	System Security (AP.2.4.3)
Quality measures category	Integrity (QM.2.1.4.1.1), Trustworthiness (QM.2.1.4)
Computing reviews category	Computer-Communications Networks Security and Protection (C.2.0), Security and Protection (K.6.5)

References and Information Sources

[Abrams 95] Abrams, Marshall D.; Jajodia, Sushil; & Podell, Harold J. *Information Security An Integrated Collection of Essays.* Los Alamitos, CA: IEEE Computer Society Press, 1995

[Schneier 96] Schneier, Bruce. *Applied Cryptography.* New York, NY: John Wiley & Sons, 1996.

[White 96] White, Gregory B.; Fisch, Eric A.; & Pooch, Udo W. *Computer System and Network Security.* Boca Raton, FL: CRC Press, 1996.

Last Modified

10 Jan 97

Object Linking and Embedding/Component Object Model

DRAFT

Note

We recommend Object Request Broker, pg. 291, as prerequisite reading for this technology description.

Purpose and Origin

OLE/COM is a specification and implementation developed by Microsoft Corporation for providing a distributed compound document framework for desktop applications. OLE/COM attempts to provide a way to simplify complexity in software, reduce the learning curve, allow interoperation and integration of multiple vendor programs, and save investment in legacy hardware/software.

Technical Detail

OLE/COM provides a standard application programming interface (API) to allow exchanging of objects and creation of standard objects for use in integrating custom applications, packaged software, or to allow diverse objects to freely interact— it is an object-enabling capability. APIs are defined at the binary level to remove language bias and promote interoperability and portability.

The OLE/COM model defines a powerful way to create documents from multiple sources from different applications. OLE objects associate two types of data: presentation data (display) and information for editing. OLE "linking" places presentation data and pointers to native data in a document while OLE "embedding" places both physically into a document. A "container" is a document with objects from multiple sources.

COM (Component Object Model) is the underlying object model and implementation for OLE and provides a binary standard for software-component interoperability. COM is based on object request broker (ORB) technology (see pg. 291) and is used to simplify the implementation of linking and embedding. COM contains object-based extensions to DCE (Distributed Computing Environment) RPC (remote procedure calls) (see pg. 323). A proprietary interface design language (IDL) called Object Definition Language is used for dynamic method invocation. COM IDL is based on the C programming language.

Usage Considerations

A number of issues must be evaluated when considering OLE/COM. They include

- *Platform support.* Because of its lack of support for a variety of platforms, OLE/COM is currently best used for 100% COTS end-user systems on Windows platforms.
- *Support for distributed objects.* COM does not yet support distributed objects, thereby reducing its effectiveness for a distributed

environment. Compound documents are non-distributed and single-user.

- *Stability of APIs.* OLE/COM APIs are heavily dependent upon Microsoft's C++ implementation and many changes are expected for the APIs, thus providing a potentially unstable basis for development/maintenance. The binary API does not guarantee portable language bindings (e.g., C binding is quite complex). OLE APIs are rather narrow in applicability. Short-term limitations are built into APIs, thus justifying obsolescence of legacy APIs.
- *Long-term system maintainability.* Maintainability may be an issue because of Microsoft's pattern for technology evolution: Constant technology change and single-generation backwards compatibility. Innovations in OLE/COM assume universal adaptation.

COM supports reuse by never changing previously-defined interface specifications— new specifications are created for each change.

Embedding uses more overhead but allows transferring and editing with a document. Linking is more efficient and single-instance but cannot travel with a document.

Maturity

OLE was originally a proprietary solution for Microsoft's Office products. Microsoft and third party developers provided basic linking and embedding in Windows 3.0 and 3.1, Windows NT version 3.1. OLE 1.0 was able to create pictures, text, charts, video clips, and sound annotations. OLE 2.0 provided a greater ability to interact seamlessly; visual editing; nested objects; storage-independent object links; adaptable links; OLE automation; version management; and object conversion (to allow different applications to use same object).

The COM technology is lagging the industry in distributed objects. A distributed implementation of a small subset of OLE2 was to be made available in late 1995. Network OLE, featuring a distributed COM, is planned for 1996 and will be based on the DCE RPC (see pg. 323). In the near future, CORBA (Common Object Request Broker Architecture) gateways to OLE/COM are planned to allow interoperability. The Object Management Group is considering standardizing interoperability between COM and CORBA. In early 1996, plans to standardize a mapping between CORBA and Network OLE were underway.

Costs and Limitations

Constant technology change drives up operations and maintenance costs.

Dependencies

Dependencies include remote procedure call (see pg. 323) and Distributed Computing Environment (see pg. 167).

Alternatives

Many groups are inventing their own integration technologies to be independent of Microsoft. Other infrastructure technologies that allow the distribution of processing across multiple processors include the following:

- Distributed Computing Environment (see pg. 167)
- remote procedure call (RPC) (see pg. 323)
- message oriented middleware (MOM) (see pg. 247)
- transaction processing monitor technology (see pg. 373)
- Common Object Request Broker Architecture (see pg. 107)
- two tier architectures (see pg. 381)

Complementary Technologies

An alternative is application programming interfaces (see pg. 79).

Index Categories

Name of technology	Object Linking and Embedding/Component Object Model
Application category	Software Architecture Models (AP.2.1.1), Client/Server (AP.2.1.2.1), Client/Server Communications (AP.2.2.1)
Quality measures category	Interoperability (QM.4.1), Portability (QM.4.2), Reusability (QM.4.4)
Computing reviews category	Distributed Systems (C.2.4), Object-Oriented Programming (D.1.5)

References and Information Sources

[Endrijones 94] Endrijones, Janet. "OLE: Not a Cheer But a Step to Integration." *Managing Automation 9,* 8 (August 1994): 30-2.

[Foody 96] Foody, M.A. "OLE and COM vs. CORBA." *Unix Review 14,* 4 (April 1996): 43-5.

[Mowbray 94] Mowbray, T.J. "Choosing Between OLE/COM and CORBA." *Object Magazine 4,* 7 (November/December 1994): 39-46.

Last Modified

10 Jan 97

Object-Oriented Analysis

IN REVIEW

Purpose and Origin

Object-oriented analysis (OOA) is concerned with developing software engineering requirements and specifications that expressed as a system's object model (which is composed of a population of interacting objects), as opposed to the traditional data or functional views of systems. OOA can yield the following benefits: *maintainability* through simplified mapping to the real world, which provides for less analysis effort, less complexity in system design, and easier verification by the user; *reusability* of the analysis artifacts which saves time and costs; and depending on the analysis method and programming language, *productivity* gains through direct mapping to features of object-oriented programming languages (see pg. 287) [Baudoin 96].

Technical Detail

An object is a representation of a real-life entity or abstraction. For example, objects in a flight reservation system might include: an airplane, an airline flight, an icon on a screen, or even a full screen with which a travel agent interacts. OOA specifies the structure and the behavior of the object— these comprise the requirements of that object. Different types of models are required to specify the requirements of the objects. The information or object model contains the definition of objects in the system, which includes: the object name, the object attributes, and object relationships to other objects. The behavior or state model describes the behavior of the objects in terms of the states the object exists in, the transitions allowed between objects, and the events that cause objects to change states. These models can be created and maintained using CASE tools that support representation of objects and object behavior.

OOA views the world as objects with data structures and behaviors and events that trigger operations, or object behavior changes, that change the state of objects. The idea that a system can be viewed as a population of interacting objects, each of which is an atomic bundle of data and functionality, is the foundation of object technology and provides an attractive alternative for the development of complex systems. This is a radical departure from prior methods of requirements specification, such as functional decomposition and structured analysis and design [Yourdon 79].

Usage Considerations

This technology works best when used in new development. The experiences of Hewlett-Packard in trying to recapture the requirements of legacy systems using OOA suggests that the process can be accomplished only when legacy systems are projected to be long-lived and frequently updated [Malan 95].

Maturity

Numerous OOA methods have been described since 1988. These OOA methods include: Shlaer-Mellor, Jacobson, Coad-Yourdon, and Rumbaugh [Baudoin 96]. The results of implementing these methods range from tremendous successes at AT&T Bell Labs [Kamath 93] to a mixture of successes and partial failures on other projects. AT&T Bell Labs realized benefits from OOA on a large project called the Call Attempt Data Collection System (CADCS). Additionally, they found during the development of two releases of the CADCS that use of the OOA techniques resulted in an 8% reduction in requirements specification time and a 30% reduction in requirements staff effort [Kamath 93]. Other OOA efforts have not been able to reproduce these successes for reasons such as the lack of completed pilot projects, and the lack of formal OOA training [Malan 95].

Costs and Limitations

The use of this technology requires a commitment to formal training in OOA methods. A method of training that has produced desired results is to initiate pilot projects, conduct formal classes, employ OOA mentors, and conduct team reviews to train properly both the analysis and development staff as well as the program management team [Kamath 93]. There are almost no reported successes using OOA methods on the first application without this type of training program [Kamath 93]. Projects with initial and continuing OOA training programs realize that the benefits of this technology depend upon the training and experience levels of their staffs. Purchase of CASE tools that support object-oriented methods may significantly enhance OOA— this is another cost to consider.

Alternatives

Alternative technologies that are used for developing software engineering requirements and specifications include functional decomposition, essential systems analysis, and structured analysis [Yourdon 79].

Complementary Technologies

There is a strong relationship between OOA and other object-oriented technologies (see Object-Oriented Database (pg. 279), Object-Oriented Design (pg. 283), and Object-Oriented Programming Languages (pg. 287)). This is especially true of object-oriented design— certain object-oriented methods combine particular analysis and design methods that work well together. In fact, the seamless use of objects throughout the analysis, design, and programming phases provides the greatest benefit. Use of OOA alone, without transition into OOD, would be a severely limited approach.

Combining object-oriented methods with Cleanroom (with its emphasis on rigor, formalisms, and reliability) (see pg. 95) can define a process capable of producing results that are reusable, predictable, and high-quality. Thus, object-oriented methods can be used for front-end domain

analysis and design, and Cleanroom can be used for life-cycle application engineering [Ett 96].

Index Categories

Name of technology	Object-Oriented Analysis
Application category	Define and Develop Requirements (AP.1.2.2.1), Analyze Functions (AP.1.2.1.1), Reengineering (AP.1.9.5)
Quality measures category	Maintainability (QM.3.1), Reusability (QM.4.4)
Computing reviews category	Software Engineering Requirements and Specifications (D.2.1), Software Engineering Tools and Techniques (D.2.2), Software Engineering Design (D.2.10)

References and Information Sources

[Baudoin 96] Baudoin, Claude & Hollowell, Glenn. *Realizing the Object-Oriented Lifecycle*. Englewood Cliffs, NJ: Prentice Hall, 1996.

[Embley 95] Embley, David W.; Jackson, Robert B.; & Woodfield, Scott N. "OO Systems Analysis: Is it or Isn't it?" *IEEE Software 12,* 2 (July 1995): 19-33.

[Ett 96] Ett, William. *A Guide to Integration of Object-Oriented Methods and Cleanroom Software Engineering* [online]. Available WWW <URL: http://www.asset.com/stars/loral/cleanroom/oo/guide.html> (1996).

[Kamath 93] Kamath, Y. H.; Smilan, R. E.; & Smith, J. G. "Reaping Benefits With Object-Oriented Technology." *AT&T Technical Journal 72*, 5 (September/October 1993): 14-24.

[Malan 95] Malan, R.; Coleman, D.; & Letsinger, R. "Lessons Learned from the Experiences of Leading-Edge Object Technology Projects in Hewlett-Packard," 33-46. *Proceedings of Tenth Annual Conference on Object-Oriented Programming Systems Languages and Applications*. Austin, TX, October 15-19, 1995. Palo Alto, CA: Hewlett-Packard, 1995.

[Yourdon 79] Yourdon, E. & Constantine, L. *Structured Design*. Englewood Cliffs, NJ: Prentice Hall, 1979.

Author

Mike Bray, Lockheed-Martin Ground Systems
michael.w.bray@den.mmc.com

Last Modified 10 Jan 97

Object-Oriented Database

IN REVIEW

Purpose and Origin

Object-oriented databases (OODBs) evolved from a need to support object-oriented programming and to reap the benefits, such as system *maintainability*, from applying object orientation to developing complex software systems. The first OODBs appeared in the late 1980s. Martin provides a complete list of these early OODBs [Martin 93]. OODBs are based on the object model and use the same conceptual models as object-oriented analysis (see pg. 275), object-oriented design (see pg. 283) and object-oriented programming (see pg. 287). Using the same conceptual model simplifies development; improves communication among users, analysts, and programmers; and lessens the likelihood of errors [Martin 93].

Technical Detail

OODBs are designed for the purpose of storing and sharing objects; they are a solution for persistent object handling. Persistent data are data that remain after a process is terminated.

There is no universally-acknowledged standard for OODBs. There is, however, some commonality in the architecture of the different OODBs because of three necessary components: object managers, object servers, and object stores. Applications interact with object managers, which work through object servers to gain access to object stores.

OODBs provide the following benefits:

- OODBs allow for the storage of complex data structures that can not be easily stored using conventional database technology.
- OODBs support all the persistence necessary when working with object-oriented languages.
- OODBs contain active object servers that support not only the distribution of data but also the distribution of work (in this context, relational database management systems (DBMS) have limited capabilities) [Vorwerk 94].

In addition, OODBs were designed to be well integrated with object-oriented programming languages such as C++ and Smalltalk. They use the same object model as these languages. With OODBs, the programmer deals with transient (temporary) and persistent (permanent) objects in a uniform manner. The persistent objects are in the OODB, and thus the conceptual walls between programming and database are removed. As stated earlier, the employment of a unified conceptual model greatly simplifies development [Tkach 94].

Usage Considerations

The type of database application should dictate the choice of database management technology. In general, database applications can be categorized into two different applications:

1. *Data collection applications* focus on entering data into a database and providing queries to obtain information about the data. Examples of these kinds of database applications are accounts payable, accounts receivable, order processing, and inventory control. Because these types of applications contain relatively simple data relationships and schema design, relational database management systems (RDBMs) are better suited for these applications.
2. *Information analysis applications* focus on providing the capability to navigate through and analyze large volumes of data. Examples of these applications are CAD/CAM/CAE, production planning, network planning, and financial engineering. These types of applications are very dynamic and their database schemas are very complex. This type of application requires a tightly-coupled language interface and the ability to handle the creation and evolution of schema of arbitrary complexity without a lot of programmer intervention. Object-oriented databases support these features to a great degree and are therefore better suited for the information analysis type of applications [Desanti 94].

OODBs are also used in applications handling BLOBs (binary large objects) such as images, sound, video, and unformatted text. OODBs support diverse data types rather than only the simple tables, columns and rows of relational databases.

Maturity

Claims have been made that OODBs are not used in mainstream applications, are not scalable, and represent an immature technology [Object 96]. Two examples to the contrary include the following:

- Northwest Natural Gas uses an OODB for a customer information system. The system stores service information on 400,000 customers and is accessed by 250 customer service representatives in seven district offices in the Pacific Northwest.
- Ameritech Advanced Data Services uses an OODB for a comprehensive management information system that currently includes accounting, order entry, pricing, and pre-sales support and is accessed by more than 200 people dispersed in a five state region.

Both of these applications are mainstream and represent databases well over a gigabyte in size; this highlights the fact that OODBs do work well with large numbers of users in large applications [Object 96].

Costs and Limitations

The costs of implementing OODB technology are dependent on the required platforms and numbers of licenses required. The costs of the actual OODB software are comparable to relational database management systems on similar platforms. The use of OODBs requires an educational

change among the software developers and database maintainers and requires a corporate commitment to formal training in the proper use of the OODB features.

Alternatives

An alternative is relational database management systems (RDBMs).

Index Categories

Name of technology	Object-Oriented Database
Application category	Database Design (AP.1.3.2), Database Administration (AP.1.9.1), Databases (AP.2.6)
Quality measures category	Maintainability (QM.3.1)
Computing reviews category	Database Management (H.2)

References and Information Sources

[Desanti 94] Desanti, Mike & Gomsi, Jeff. "A Comparison of Object and Relational Database Technologies." *Object Magazine 3*, 5 (January 1994): 51-57.

[Martin 93] Martin, James. *Principles of Object-Oriented Analysis and Design.* Englewood Cliffs, NJ: Prentice Hall, 1993.

[Object 96] "Focus on ODBMS Debunking the Myths." *Object Magazine 5*, 9 (February 1996): 21-23.

[Tkach 94] Tkach, Daniel & Puttick, Richard. *Object Technology in Application Development.* Redwood City, CA: The Benjamin/Cummings Publishing Company, Inc., 1994.

[Vorwerk 94] Vorwerk, Raymond. "Towards a True OBBMS." *Object Magazine 3*, 5 (January 1994): 38-39.

Author

Mike Bray, Lockheed-Martin Ground Systems
michael.w.bray@den.mmc.com

Last Modified

10 Jan 97

Object-Oriented Design

IN REVIEW

Purpose and Origin

Object-oriented design (OOD) is concerned with developing an object-oriented model of a software system to implement the identified requirements. Many OOD methods have been described since the late 1980s. The most popular OOD methods include Booch, Buhr, Wasserman, and the HOOD method developed by the European Space Agency [Baudoin 96]. OOD can yield the following benefits: *maintainability* through simplified mapping to the problem domain, which provides for less analysis effort, less complexity in system design, and easier verification by the user; *reusability* of the design artifacts, which saves time and costs; and productivity gains through direct mapping to features of object-oriented programming languages (see pg. 287) [Baudoin 96].

Technical Detail

OOD builds on the products developed during object-oriented analysis (OOA) (see pg. 275) by refining candidate objects into classes, defining message protocols for all objects, defining data structures and procedures, and mapping these into an object-oriented programming language (OOPL) (see pg. 287). Several OOD methods (Booch, Shlaer-Mellor, Buhr, Rumbaugh) describe these operations on objects, although none is an accepted industry standard. Analysis and design are closer to each other in the object-oriented approach than in structured analysis and design. For this reason, similar notations are often used during analysis and the early stages of design. However, OOD requires the specification of concepts nonexistent in analysis, such as the types of the attributes of a class, or the logic of its methods.

Design can be thought of in two phases. The first, called high-level design, deals with the decomposition of the system into large, complex objects. The second phase is called low-level design. In this phase, attributes and methods are specified at the level of individual objects. This is also where a project can realize most of the reuse of object-oriented products, since it is possible to guide the design so that lower-level objects correspond exactly to those in existing object libraries or to develop objects with reuse potential. As in OOA, the OOD artifacts are represented using CASE tools with object-oriented terminology.

Usage Considerations

OOD techniques are useful for development of large complex systems. AT&T Bell Labs used OOD and realized the benefits of reduced product development time and increased reuse of both code and analysis/design artifacts on a large project called the Call Attempt Data Collection System (CADCS). This large project consisted of over 350,000 lines of C++ code that ran on a central processor with over 100 remote systems distributed across the United States. During the development of two releases of the

CADCS they found that use of the OOD techniques resulted in a 30% reduction in development time and a 20% reduction in development staff effort as compared to similarly sized projects using traditional software development techniques [Kamath 93]. However, these successes were realized only after thorough training and completion of three- to six-month pilot projects by their development staff [Kamath 93].

Experiences from other organizations show costly learning curves and few productivity improvements without thoroughly-trained designers and developers. Additionally, OOD methods must be adapted to the project since each method contains object models that may be too costly, or provide little value, for use on a specific project [Malan 95].

The maximum impact from OOD is achieved when used with the goal of designing reusable software systems. For objects without significant reuse potential, OOD techniques were more costly than traditional software development methodologies. This was because of the costs associated with developing objects and the software to implement these objects for a one-time use [Maring 96].

Maturity

Many OOD methods have been used in industry since the late 1980s. OOD has been used worldwide in many commercial, Department of Defense (DoD), and government applications. There exists a wealth of documentation and training courses for each of the various OOD methods, along with commercially-available CASE tools with object-oriented extensions that support these OOD methods.

Costs and Limitations

One reason for the mixed success reviews on OOD techniques is that the use of this technology requires a corporate commitment to formal training in the OOD methods and the purchase of CASE tools with capabilities that support these methods. The method of training that produces the best results is to initiate pilot projects, conduct formal classes, employ OOD mentors, and conduct team reviews to train properly both the analysis and development staff as well as the program management team [Kamath 93]. There are few, if any, reported successes using OOD methods on the first application without this type of training program [Maring 96]. Projects with initial and continuing OOD training programs realize that the benefits of this technology depend upon the training and experience levels of their staffs.

Dependencies

The use of OOD technology requires the development of object requirements using OOA techniques, and CASE tools to support both the drawing of objects and the description of the relationships between objects. Also, the final steps of OOD, representing classes and objects in programming constructs, are dependent on the object-oriented program-

ming language (OOPL) (see pg. 287) chosen. For example, if the OOPL is Ada 95 (see pg. 67), a package-based view of the implementation should be used; if C++ is the OOPL, then a class-based view should be used. These different views require different technical design decisions and implementation considerations.

Alternatives

An alternative technology that can be used for developing a model of a software system design to implement the identified requirements is a traditional design approach such as Yourdon and Constantine's Structured Design [Yourdon 79]. This method, used successfully for many different types of applications, is centered around design of the required functions of a system and does not lend itself to object orientation.

Complementary Technologies

Combining object-oriented methods with Cleanroom (with its emphasis on rigor, formalisms, and reliability) (see pg. 95) can define a process capable of producing results that are reusable, predictable, and high-quality. Thus, object-oriented methods can be used for front-end domain analysis and design, and Cleanroom can be used for life-cycle application engineering [Ett 96].

Index Categories

Name of technology	Object-Oriented Design
Application category	Detailed Design (AP.1.3.5), Reengineering (AP.1.9.5)
Quality measures category	Maintainability (QM.3.1), Reusability (QM.4.4)
Computing reviews category	Object-Oriented Programming (D.1.5), Software Engineering Design (D.2.10)

References and Information Sources

[Baudoin 96] Baudoin, Claude & Hollowell, Glenn. *Realizing the Object-Oriented Lifecycle.* Upper Saddle River, NJ: Prentice Hall, 1996.

[Ett 96] Ett, William. *A Guide to Integration of Object-Oriented Methods and Cleanroom Software Engineering* [online]. Available WWW <URL: http://www.asset.com/stars/loral/cleanroom/oo/guide.html> (1996).

[Kamath 93] Kamath, Y. H.; Smilan, R. E.; & Smith, J. G. "Reaping Benefits With Object-Oriented Technology." *AT&T Technical Journal 72,* 5 (September 1993): 14-24.

[Malan 95] Malan, R.; Coleman, D.; & Letsinger, R. "Lessons Learned from the Experiences of Leading-Edge Object Technology Projects in Hewlett-Packard," 33-46. *Proceedings of Tenth Annual Conference on Object-*

Oriented Programming Systems Languages and Applications. Austin, TX, October 15-19, 1995. Palo Alto, CA: Hewlett-Packard, 1995.

[Maring 96] Maring, B. "Object-Oriented Development of Large Applications." *IEEE Software 13,* 3 (May 1996): 33-40.

[Yourdon 79] Yourdon, E. & Constantine, L. *Structured Design*. Englewood Cliffs, NJ: Prentice Hall, 1979.

Author Mike Bray, Lockheed-Martin Ground Systems
michael.w.bray@den.mmc.com

Last Modified 10 Jan 97

Object-Oriented Programming Languages — IN REVIEW

Purpose and Origin

Object-oriented programming languages (OOPLs) are the natural choice for implementation of an object-oriented design (see pg. 283) because they directly support the object notions of classes, inheritance, information hiding, and dynamic binding. Because they support these object notions, OOPLs make an object-oriented design easier to implement [Baudoin 96]. An object-oriented system programmed with an OOPL results in less complexity in the system design and implementation, which can lead to an increase in *maintainability* [Baudoin 96]. The genesis of this technology dates back to the early 1960s with the work of Nygaard and Dahl in the development of the first object-oriented language called Simula 67. Research progressed through the 1970s with the development of Smalltalk at Xerox. Current OOPLs include C++, Objective C, Smalltalk, Eiffel, Common LISP Object System (CLOS), Object Pascal, Java (see pg. 221), and Ada 95 (see pg. 67) [Baudoin 96].

Technical Detail

Object-oriented (OO) applications can be written in either conventional languages or OOPLs, but they are much easier to write in languages especially designed for OO programming. OO language experts divide OOPLs into two categories, hybrid languages and pure OO languages. Hybrid languages are based on some non-OO model that has been enhanced with OO concepts. C++ (a superset of C), Ada 95, and CLOS (an object-enhanced version of LISP) are hybrid languages. Pure OO languages are based entirely on OO principles; Smalltalk, Eiffel, Java, and Simula are pure OO languages.

In terms of numbers of applications, the most popular OO language in use is C++. One advantage of C++ for commercial use is its syntactical familiarity to C, which many programmers already know and use; this lowers training costs. Additionally, C++ implements all the concepts of object orientation, which include classes, inheritance, information hiding, polymorphism, and dynamic binding. One disadvantage of C++ is that it lacks the level of polymorphism and dynamics most OO programmers expect. Ada 95 is a reliable, standardized language well-suited for developing large, complex systems that are reliable [Tokar 96].

The major alternative to C++ or Ada 95 is Smalltalk. Its advantages are its consistency and flexibility. Its disadvantages are its unfamiliarity (causing an added training cost for developers), and its inability to work with existing systems (a major benefit of C++) [Tokar 96].

Usage Considerations

OOPLs are strongly recommended to complete the implementation of object-oriented analysis (OOA) (see pg. 275) and object-oriented design

(OOD) (see pg. 283) technologies. AT&T Bell Labs used OOD and OOPLs and realized the benefits of reduced product development time and increased reuse of both code and analysis/design artifacts on a large project called Call Attempt Data Collection System (CADCS). This large project consisted of over 350,000 lines of C++ code that ran on a central processor with over 100 remote systems distributed across the United States. During the development of two releases of the CADCS, the use of the OOD techniques and subsequent implementation in OOPL resulted in an 30% reduction in development time and a 20% reduction in development staff effort as compared to similarly-sized projects using traditional software development techniques and languages [Kamath 93].

Organizations such as Bell Labs have found that through the introduction of OO programming techniques in pilots and training courses, the developers were able to learn properly and experiment with the OOPL constructs. This resulted in increased object-oriented expertise such that much of the CADCS software (objects) was reused on a similar project [Kamath 93].

OOPLs such as Ada 95 and C++ can also be used to develop traditional non-object-oriented software. These applications can be developed by avoiding the use of the object-oriented language features. There are many commercial, Department of Defense (DoD), and government applications of this type in existence today.

For applications where OOPL code is to be generated by a CASE tool, developers must decide which programming language to generate: C++, Ada 95, Smalltalk, Java, or CLOS. The choice of an OOPL can limit the choices of CASE tools because the tools may not support the chosen language. However, if language generation is not a consideration, then CASE tools can be chosen based on features and design capabilities without regard to the OOPL chosen for implementation.

Since different OOPLs support different levels of 'objectiveness' (e.g., inheritance), different OOD constructs may or may not map directly to OOPL constructs. Therefore, the choice of an OOPL is affected by a design captured using OOD techniques. Where OOD is not present, any OOPL can be used, depending upon the training of the developers.

Maturity

OOPLs have been used worldwide on many commercial, DoD, and government applications/projects. There exists a wealth of documentation and training courses for each of the various OOPLs.

Costs and Limitations

The use of OOPL technology requires a corporate commitment to formal training in the proper use of the OOPL features and the purchase of the language compiler. The costs of completely training a development staff implies that the insertion of this technology should be undertaken only on new developments (instead of maintenance of legacy systems), and only after pilot project(s) are successfully completed [Malan 95].

Alternatives

Both object-oriented and non-object-oriented applications can be written in either traditional languages or OOPLs. To fully realize the benefits of an object orientation, it is much easier to write the implementations in languages especially designed for OO programming.

Complementary Technologies

Combining object-oriented methods with Cleanroom (with its emphasis on rigor, formalisms, and reliability) (see pg. 95) can define a process capable of producing results that are reusable, predictable, and high-quality. Thus, OOPLs can be used for implementation of an object-oriented design and Cleanroom can be used for life-cycle application engineering.

Index Categories

Name of technology	Object-Oriented Programming Languages
Application category	Programming Language (AP.1.4.2.1)
Quality measures category	Maintainability (QM.3.1)
Computing reviews category	Object-Oriented Programming (D.1.5), Programming Language Classifications (D.3.2)

References and Information Sources

[Baudoin 96] Baudoin, Claude & Hollowell, Glenn. *Realizing the Object-Oriented Lifecycle.* Upper Saddle River, NJ: Prentice Hall, 1996.

[Kamath 93] Kamath, Y. H.; Smilan, R. E.; & Smith, J. G. "Reaping Benefits with Object-Oriented Technology." *AT&T Technical Journal 72,* 5 (September 1993): 14-24.

[Malan 95] Malan, R.; Coleman, D.; & Letsinger, R. "Lessons Learned from the Experiences of Leading-Edge Object Technology Projects in Hewlett-Packard," 33-46. *Proceedings of Tenth Annual Conference on Object-Oriented Programming Systems Languages and Applications.* Austin, TX, October 15-19, 1995. Palo Alto, CA: Hewlett-Packard, 1995.

[Tokar 96] Tokar, Joyce L. "Ada 95: The Language for the 90's and Beyond." *Object Magazine 6,* 4 (June 1996): 53-56.

Author

Mike Bray, Lockheed-Martin Ground Systems
michael.w.bray@den.mmc.com

Last Modified 10 Jan 97

Object Request Broker

COMPLETE

Note

We recommend Middleware, pg. 251, as prerequisite reading for this technology description.

Purpose and Origin

An object request broker (ORB) is a middleware technology that manages communication and data exchange between objects. ORBs promote *interoperability* of distributed object systems because they enable users to build systems by piecing together objects— from different vendors— that communicate with each other via the ORB [Wade 94]. The implementation details of the ORB are generally not important to developers building distributed systems. The developers are only concerned with the object interface details. This form of information hiding enhances system *maintainability* since the object communication details are hidden from the developers and isolated in the ORB [Cobb 95].

Technical Detail

ORB technology promotes the goal of object communication across machine, software, and vendor boundaries. The relevant functions of an ORB technology are

- interface definition
- location and possible activation of remote objects
- communication between clients and object

An object request broker acts as a kind of telephone exchange. It provides a directory of services and helps establish connections between clients and these services [CORBA 96, Steinke 95]. Figure 21 illustrates some of the key ideas.

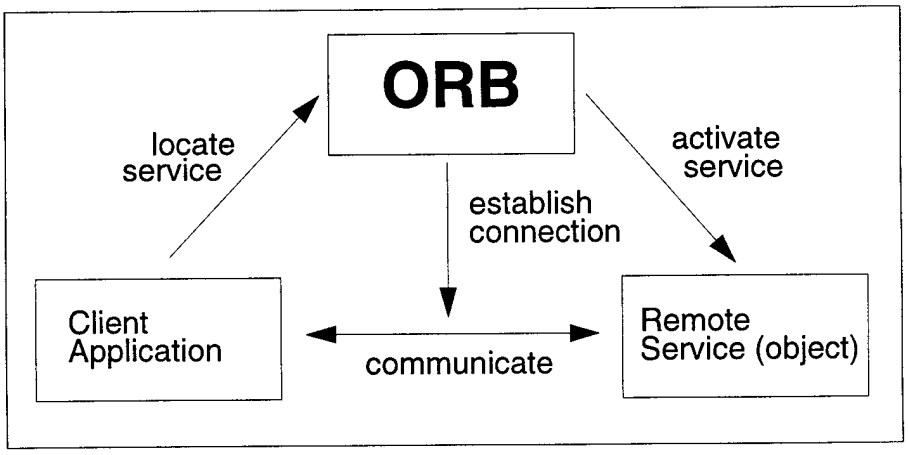


Figure 21: Object Request Broker

Note that there are many ways of implementing the basic ORB concept; for example, ORB functions can be compiled into clients, can be separate processes, or can be part of an operating system kernel. These basic design decisions might be fixed in a single product, as is the case with Microsoft's OLE (see below); or there might be a range of choices left to the ORB implementer, as is the case with CORBA (see below).

The ORB must support many functions in order to operate consistently and effectively, but many of these functions are hidden from the user of the ORB. It is the responsibility of the ORB to provide the illusion of locality, in other words, to make it appear as if the object is local to the client, while in reality it may reside in a different process or machine [Reddy 95]. Thus the ORB provides a framework for cross-system communication between objects. This is the first technical step toward interoperability of object systems.

The next technical step toward object system interoperability is the communication of objects across platforms. An ORB allows objects to hide their implementation details from clients. This can include programming language, operating system, host hardware, and object location. Each of these can be thought of as a "transparency,"[1] and different ORB technologies may choose to support different transparencies, thus extending the benefits of object orientation across platforms and communication channels.

There are two major ORB technologies:

- The Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) specification (see pg. 107)
- Microsoft's Common Object Model, partly implemented in Object Linking and Embedding (OLE) (see pg. 271)

An additional, newly-emerging ORB model is Remote Method Invocation (RMI); this is specified as part of the Java language/virtual machine (see pg. 221). RMI allows Java objects to be executed remotely. This provides ORB-like capabilities as a native extension of Java.

Usage Considerations

Successful adoption of ORB technology requires a careful analysis of the current and future software architectural needs of the target application and analysis of how a particular ORB will satisfy those needs [Abowd 96]. Among the many things to consider are platform availability, support for various programming languages, and product performance parameters.

1. transparency: making something invisible to the client

After performing this analysis, developers can make informed decisions in choosing the ORB best suited for their application's needs. A quick reference chart on ORBs follows:

ORB	Platform Availability	Applicable to	Mechanism	Implementations
OLE/ COM	PC platforms	document management architecture	APIs to proprietary system	one
CORBA	platform-independent and interoperability among platforms	general distributed system architecture	specification of distributed object technology	many[a]
Java/ RMI	wherever Java virtual machine (VM) executes	general distributed system architecture and Web-based Intranets	implementation of distributed object technology	various[b]

a. Examples include ORBIX by IONA Technology, NEO by SunSoft, VisiBroker by VisiGenic, PowerBroker by Expersoft, SmallTalkBroker by DNS Technologies, Object Director by Fujitsu, DSOM by IBM, DAIS by ICL, SORBET by Siemens Nixdorf, and NonStop DOM by Tandem.

b. Implementations of the Java VM have been ported to various platforms.

Maturity

As shown in the chart above, there are a number of commercial ORB products available. ORB products that are not compliant with either CORBA or OLE also exist; however, these tend to be vendor-unique solutions that may affect system interoperability, portability, and maintainability.

Major developments in commercial ORB products are occurring, with life cycles seemingly lasting only four to six months. In addition, new ORB technology (Java/RMI) is emerging, and there are signs of potential "mergers" involving two of the major technologies. The continued trend toward Intranet- and Internet-based applications is another stimulant in the situation. Whether these commercial directions are fully technically viable and will be accepted by the market is unknown.

Given the current situation and technical uncertainty, potential users of ORB technologies need to determine

- what new features ORB technologies add beyond technologies currently in use in their organizations
- the potential benefits from using these new features
- the key risks involved in adopting the technology as a whole
- how much risk is acceptable to them

One possible path would be to undertake a disciplined and "situated" technology evaluation. Such an evaluation, as described by Brown and Wallnau, focuses on evaluating so-called "innovative" technologies and can provide technical information for adoption that is relative to the current/existing approaches in use by an organization [Brown 96, Wallnau 96]. Such a technology evaluation could include pilot projects focusing on model problems pertinent to the individual organization.

Costs and Limitations

The license costs of the ORB products from the vendors listed above are dependent on the required operating systems and the types of platform. ORB products are available for all major computing platforms and operating systems.

Index Categories

Name of technology	Object Request Broker
Application category	Client/Server (AP.2.1.2.1), Client/Server Communication (AP.2.2.1)
Quality measures category	Interoperability (QM.4.1), Maintainability (QM.3.1)
Computing reviews category	Distributed Systems (C.2.4), Object-Oriented Programming (D.1.5)

References and Information Sources

[Abowd 96] Abowd, Gregory, et al. "Architectural Analysis of ORBs." *Object Magazine 6*, 1 (March 1996): 44-51.

[Brown 96] Brown, A. & Wallnau, Kurt. "A Framework for Evaluating Software Technology." *IEEE Software 13,* 5 (September 1996): 39-49.

[Cobb 95] Cobb, Edward E. "TP Monitors and ORBs: A Superior Client/Server Alternative." *Object Magazine 4*, 9 (February 1995): 57-61.

[CORBA 96] *The Common Object Request Broker: Architecture and Specification*, Version 2.0. Framingham, MA: Object Management Group, 1996. Also available [online] WWW <URL: http://www.omg.org> (1996).

[Reddy 95] Reddy, Madhu. "ORBs and ODBMSs: Two Complementary Ways to Distribute Objects." *Object Magazine 5*, 3 (June 1995): 24-30.

[Steinke 95] Steinke, Steve. "Middleware Meets the Network." *LAN Magazine 10,* 13 (December 1995): 56.

[Tkach 94] Tkach, Daniel & Puttick, Richard. *Object Technology in Application Development.* Redwood City, CA: Benjamin/Cummings Publishing Company, 1994.

[Wade 94] Wade, Andrew E. "Distributed Client-Server Databases." *Object Magazine 4,* 1 (April 1994): 47-52.

[Wallnau 96] Wallnau, Kurt & Wallace, Evan. "A Situated Evaluation of the Object Management Group's (OMG) Object Management Architecture (OMA)," 168-178. *Proceedings of the OOPSLA'96.* San Jose, CA, October 6-10, 1996. New York, NY: ACM, 1996. Also available [online] FTP. <URL: ftp://ftp.sei.cmu.edu/pub/corba/OOPSLA/present> (1996).

Authors Kurt Wallnau, SEI
kcw@sei.cmu.edu

John Foreman, SEI
jtf@sei.cmu.edu

Mike Bray, Lockheed-Martin Ground Systems
michael.w.bray@den.mmc.com

External Reviewer(s) Ed Morris, SEI
Richard Soley, VP, Chief Technical Officer, Object Management Group

Last Modified 10 Jan 97

Organization Domain Modeling

COMPLETE

Note

We recommend Domain Engineering and Domain Analysis, pg. 173, as prerequisite reading for this technology description.

Purpose and Origin

Organization domain modeling (ODM) was developed to provide a formal, manageable, and repeatable approach to domain engineering. The ODM method evolved and was subsequently formalized by Mark Simos (Organon Motives, Inc.) with collaboration and sponsorship from Hewlett-Packard Company, Lockheed-Martin,[1] and the DARPA STARS[2] program [Simos 96]. ODM affects the *maintainability*, *understandability*, and *reusability* characteristics of a system or family of systems.

Technical Detail

ODM was developed and refined as part of the overall reuse/product line approaches developed under the STARS program. The STARS reuse approach decomposes reuse technologies into several levels or layers of abstraction, specifically: Concepts, Processes, Methods, and Tools. An example of a "concept" is the Conceptual Framework for Reuse Processes (CFRP), a conceptual foundation and framework for understanding domain-specific reuse in terms of the processes involved [STARS 93]. An example of a "process" is the Reuse-Oriented Software Evolution (ROSE) process model, which is based on the CFRP life-cycle process model; it partitions software development into domain engineering, asset management, and application engineering; and emphasizes the role of reuse in software evolution. ODM is an example of a "method" compatible with the CFRP framework.

The primary goal of ODM is the systematic transformation of artifacts (e.g., requirements, design, code, tests, and processes) from multiple legacy systems into assets that can be used in multiple systems. The method can also be applied to requirements for new systems; the key element is to ground domain models empirically by explicit consideration of multiple exemplars, which determine the requisite range of variability that the models must encompass. ODM stresses the use of legacy artifacts and knowledge as a source of domain knowledge and potential resources for reengineering/reuse. However, one of its objectives is to avoid embedding hidden constraints that may exist in legacy systems into the domain models and assets.

---

1. formerly Unisys Defense Systems, Reston, VA

2. Defense Advanced Research Projects Agency (DARPA) Software Technology for Adaptable, Reliable Systems (STARS)

Domain Engineering and Domain Analysis, pg. 173, identifies three areas where domain analysis methods can be differentiated. Distinguishing features for ODM are:

Primary product of the analysis. The result of ODM is a knowledge representation framework populated with a domain architecture and a flexible asset base. It can be thought of as a reuse library designed to support systematic reuse in a prescribed context; however, the method supports the use of diverse implementation techniques such as generators in the asset base.

Focus of analysis

- ODM is structured in terms of a core domain engineering life cycle, which is distinct from and orthogonal to the system engineering life cycle. The ODM life cycle is divided into three phases:
  - plan domain: selecting, scoping, and defining target domains
  - model domain: modeling the range of variability that can exist within the scope of the domain
  - engineer asset base: engineering an asset base that satisfies some subset of the domain variability, based on the needs of specific target customers [Simos 96]
- *Iterative scoping.* The approach to systematic scoping involves structuring the ODM life cycle as a series of incremental scoping steps; each step builds upon and validates the previous step.
- *Stakeholder focus.* The ODM life cycle provides an up-front analysis of the organizational stakeholders and objectives. The stakeholder focus is carried throughout the life cycle with tasks to reconsider the strategic interests of stakeholders at critical points.
- *Exemplar-based modeling.* ODM works from a set of explicit examples, called exemplars, of the domain rather than a single, generalized example or speculation about a "general" solution.
- *Emphasis on descriptive modeling.* ODM places heavy emphasis on studying a set of example systems for the domain in order to derive the shape of the domain space.
- *Explicit modeling of variability.* ODM encourages modelers to maximize variability in the descriptive phase of modeling. This is to generate as much insight as possible about the potential range of variability in the domain.
- *Methods for context recovery.* ODM emphasizes identifying contextual information (e.g., language, values, assumptions, dependencies, history) embedded within an artifact to make them more dependable and predictable. (Note: This activity does not

remove dependencies. This occurs during the engineering asset base phase.)

- *Prescriptive asset base engineering.* After descriptive modeling takes place, the prescriptive modeling phase begins. Initially, the range of functionality to be supported by the reusable assets are re-scoped and commitments are made to a real set of customers. Prescriptive features are mapped onto the structure of the asset base and to sets of specifications for particular assets. Traceability from the features back to exemplar artifacts are maintained to enable the retrieval of additional information (e.g., existing prototypes, history).

Representation Techniques. Although ODM encompasses all of domain engineering, the core method focuses on activities that are unique to domain engineering. Other activities that fall within, but are not specific to domain engineering are supported through "supporting methods." This means that ODM can be integrated with a variety of existing methods (e.g., system and taxonomic modeling techniques) to support unique constraints or preferences of an organization or domain. Examples of supporting methods are the methods associated with the Reuse Library Framework (RLF) [STARS 96c], Canvas [STARS 96a], Domain Architecture-Based Generation for Ada Reuse (DAGAR) [Klinger 96, STARS 96b], and the Knowledge Acquisition for Preservation of Tradeoffs and Underlying Rationales (KAPTUR) Tool, which is described as a part of Argument-Based Design Rationale Capture Methods for Requirements Tracing, pg. 91.

Usage Considerations

ODM was developed primarily to support domain engineering projects for domains that are mature, reasonably stable, and economically viable. Although all of the criteria do not need to be met, ODM is most successful when all are present. ODM can be applied in reuse programs that are in their infancy or very mature. ODM does not assume application within an established reuse program, and in fact includes some risk-reduction steps (such as up-front stakeholder analysis) that enable the use of domain analysis as a first step in establishing such a program.

However, it is recommended that the first application of ODM be on a pilot project in a relatively small domain. ODM supports evolution to larger domains or a broader reuse program.

Maturity

ODM has been applied on small-scale and relatively large-scale projects. The following are examples:

- Hewlett-Packard developed a domain engineering workbook by tailoring aspects of an early version of the ODM process model to their organizational objectives. The workbook is being used on

numerous internal domain engineering projects within their divisions [Cornwell 96, Simos 96].

- The Air Force CARDS Program applied ODM in several different areas: as a means of structuring a comparative study on architecture representation languages; on the automated message handling system (AMHS) domain analysis effort; and for product-line analysis as part of the Hanscom AFB Domain Scoping effort.
- ODM formed the basis for the domain engineering approach of the Army STARS Demonstration Project. ODM processes were integrated closely with the CFRP [STARS 93] as a higher level planning guide, and with RLF as a domain modeling representation technology [Lettes 96].

Costs and Limitations

Before incorporating ODM into the overall reuse plan, an organization should consider the following:

- The core ODM method does not directly address the ongoing management of domain models and assets, or the use of the assets by application development projects. These activities are part of a larger reuse program described in the CFRP [STARS 93].
- ODM does not encompass the overall reuse program planning including the establishment of producer-consumer relationships between domain engineering projects and other efforts, such as system reengineering projects or planned new projects.
- ODM may not be applicable within organizations that are not prepared to commit to, or at least experiment with, systematic reuse (i.e., reuse of assets that were developed using a software engineering process that is specifically structured for reuse).
- ODM requires that an organization adopt the level of modeling rigor, the modeling styles, or approaches recommended within ODM.
- The use of ODM necessitates a technology infrastructure and level of technical expertise sufficient to support ODM modeling needs [Simos 96].

Complementary Technologies

A complimentary technology is generation techniques.

Index Categories

Name of technology	Organization Domain Modeling
Application category	Domain Engineering (AP.1.2.4)
Quality measures category	Reusability (QM.4.4), Maintainability (QM.3.1), Understandability (QM.3.2)
Computing reviews category	Software Engineering Tools and Techniques (D.2.2)

References and Information Sources

[Cornwell 96] Cornwell, Patricia Collins. "HP Domain Analysis: Producing Useful Models for Reusable Software." *HP Journal* (August 1996): 46-55.

[Klinger 96] Klinger, Carol & Solderitsch, James. *DAGAR: A Process for Domain Architecture Definition and Asset Implementation* [online]. Available WWW <URL: http://source.asset.com/stars/darpa/Papers/ArchPapers.html> (1996).

[Lettes 96] Lettes, Judith A. & Wilson, John. *Army STARS Demonstration Project Experience Report* (STARS-VC-A011/003/02). Manassas, VA: Loral Defense Systems-East, 1996.

[Simos 94] Simos, M. "Juggling in Free Fall: Uncertainty Management Aspects of Domain Analysis Methods," 512-521. *Fifth International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems*. Paris, France, July 4-8, 1994. Berlin, Germany: Springer-Verlag, 1995.

[Simos 96] Simos, M., et al. *Software Technology for Adaptable Reliable Systems (STARS). Organization Domain Modeling (ODM) Guidebook* Version 2.0 (STARS-VC-A025/001/00). Manassas, VA: Lockheed Martin Tactical Defense Systems, 1996. Also available [online] WWW <URL: http://www.asset.com/WSRD/abstracts/ABSTRACT_1176.html> (1996).

[STARS 93] *Conceptual Framework for Reuse Processes* Volume I, Definition, Version 3.0 (STARS-VC-A018/001/00). Reston, VA: Software Technology for Adaptable Reliable Systems, 1993.

[STARS 96a] *Canvas Knowledge Acquisition Guide Book* Version 1.0 (STARS-PA29-AC01/001/00) Reston, VA: Software Technology for Adaptable, Reliable Systems, 1996.

[STARS 96b] *Domain Architecture-Based Generation for Ada Reuse (DAGAR) Guidebook* Version 1.0. Manassas, VA: Lockheed Martin Tactical Defense Systems, 1996.

[STARS 96c] *Open RLF* (STARS-PA31-AE08/001/00). Manassas, VA: Lockheed Martin Tactical Defense Systems, 1996.

Author

Liz Kean, Rome Laboratory
liz@se.rl.af.mil

External Reviewer(s)	Dick Creps, Lockheed Martin, Manassas, VA Teri Payton, Lockheed Martin, Manassas, VA Mark Simos, Organon Motives, Inc., Belmont, MA
Last Modified	10 Jan 97

Personal Software Process for Module-Level Development

COMPLETE

Purpose and Origin

Personal Software Process (PSP)[1] is a framework of advanced process and quality techniques to help software engineers improve their performance and that of their organizations through a step-by-step, disciplined approach to measuring and analyzing their work. Software engineers that use the PSP can substantially improve their ability to estimate and plan their work and significantly improve the quality, i.e., reduce the defects, in the code they develop. PSP is a result of research by Watts Humphrey into applying process principles to the work of individual software engineers and small software teams [Humphrey 95]. The objective was to transfer the quality concepts of the Capability Maturity Model (CMM)[2] for Software [Paulk 95] to the individual and small team level.

Technical Detail

The foundations of PSP are the advanced process and quality methods that have been used in manufacturing to improve all forms of production. These concepts include the following:

- definition of the processes
- use of the defined processes
- measurement of the effects of the processes on product
- analysis of the effects on the product
- continuous refinement of the processes based on analysis

Some of the engineering methods used in PSP are data gathering, size and resource estimating, defect management, yield management, and cost of quality and productivity analysis. The basic data gathered in PSP are

- the time the engineer spends in each process phase
- the defects introduced and found in each phase
- the size of the developed product

All PSP process quality measures are derived from this basic set of data. Size and resource estimating is done using a proxy-based estimating method, PROBE [Humphrey 96b], that uses the engineer's personal data

1. Personal Software Process and PSP are service marks of Carnegie Mellon University.

2. Capability Maturity Model and CMM are service marks of Carnegie Mellon University.

and statistical techniques to calculate a new item's most likely size and development time, and the likely range of these estimates. A key PSP tenet is that defect management is an engineer's personal responsibility. By analyzing the data gathered on the defects they injected, engineers refine their personal process to minimize injecting defects, and devise tailored checklists and use personal reviews to remove as many defects as possible. Yield, the principal PSP quality measure, is used to measure the effectiveness of review phases. Yield is defined as the percentage of the defects in the product at the start of the review that were found during the review phase. The basic cost-of-quality measure in PSP is the appraisal-to-failure-ratio (A/FR), which is the ratio of the cost of the review and evaluation phases to the cost of the diagnosing and repair phases. PSP-trained engineers learn to relate productivity and quality, i.e., that defect-free (or nearly so) products require much less time in diagnosing and repair phases, so their projects will likely be more productive.

The PSP (and the courses based on it) concentrates on applying PSP to the design, code, and Unit Test phases, i.e. module-level development phases of software development [Humphrey 95]. As such, an instance of PSP for module-level development is created. In PSP for module-level development

- The individual process phases are planning, design, design review, code, code review, compile, test, and postmortem.
- Size is measured in lines of code and size/resource estimating is done using functions or objects as the proxies for the PROBE method.
- Engineers build individually tailored checklists for design review and code review based on the history of the defects they inject most frequently.
- Yield is the percentage of defects found before the first compile, engineers are taught to strive for a 100% yield and can routinely achieve yields in the range of 70%.
- A/FR is the ratio of the time spent in design review and code review phases to the time spent in compile and test phases, and engineers are encouraged to plan for an A/FR of 2 or higher, which has been shown to correlate well with high yield.

The PSP for module-level development substantially improves engineering performance on estimating accuracy, defect injection, and defect removal. Class data from 104 engineers that have taken the PSP course shows reductions of 58% in the average number of defects injected (and found in development) per 1,000 lines of code (KLOC), and reductions of 72% in the average number of defects per KLOC found in test. Estimating and planning accuracy also improved with an average 26% reduction in

size estimating error and an average 46% reduction in time estimating error. Average productivity of the group went up slightly [Humphrey 96a].

Usage Considerations

PSP for module-level development is applicable to new development and enhancement work on whole systems or major subunits. Based on use and analysis to date, introduction of PSP for module-level development is most effective in software organizations that have launched process improvement work and are implementing CMM Level 2 [Paulk 95] or higher practices. Because PSP emphasizes early defect removal, i.e., spending time in the design through code review phases to prevent and remove as many defects as possible before the first compile, PSP introduction will likely be difficult in organizations that are concerned primarily with schedule and not with product quality.

PSP is an individual development process; however, it can be used by small teams if all members are PSP-trained and team conventions are established for code counting and defect types. PSP does not require sophisticated tools or software development environments; however, simple spreadsheet-based tools can significantly help individual engineers reduce the effort needed to track and analyze their personal data.

The principles of PSP are being applied to other areas such as developing documentation, handling maintenance, conducting testing, and doing requirements analysis. Humphrey describes how the PSP can be adapted to these and other areas [Humphrey 95].

Maturity

PSP is a new technology. It is already being taught in a number of universities, but industrial introduction has just begun and only limited results are available. Early industrial results are similar to the PSP course results, showing reduced defects and improved quality. Work on industrial transition methods, support tools, and operational practices is ongoing.

Costs and Limitations

PSP training (class room instruction, programming exercises, and personal data analysis reports) requires approximately 150 to 200 hours on the part of each engineer. Through the use of 10 programming exercises and 5 reports, engineers are led through a progressive sequence of software processes, taking them from their current process to the full-up PSP for module-level development process. By doing the exercises, engineers learn to use the methods and by analyzing their own data, engineers see how the methods work for them.

Based on demonstrated benefits from course data, it is projected that the costs of training will be recovered by an organization within one year

through reduced defects, reduced testing time, improved cycle time, and improved product quality.

Attempts by engineers to learn PSP methods by reading the book and then trying to apply the techniques on real projects have generally not worked. Until they have practiced PSP methods and have become convinced of their effectiveness, engineers are not likely to apply them on the job.

PSP introduction requires strong management commitment because of the significant effort required to learn PSP. Management must provide engineers the time to learn PSP and track their training progress to ensure the training is completed.

Complementary Technologies

PSP is complementary to organizational software process improvements efforts based on the CMM for Software [Paulk 95]. The CMM is an organization-focused process-improvement framework that provides a disciplined, efficient organizational environment for software engineering work. The PSP equips engineers with the personal skills and methods to do high-quality work and participate in organizational process improvement. Of the 18 key process areas in the CMM, PSP demonstrates/covers 12 of the 18.

Index Categories

Name of technology	Personal Software Process for Module-Level Development
Application category	Detailed Design (AP.1.3.5), Code (AP.1.4.2), Unit Testing (AP.1.4.3.4), Component Testing (AP.1.4.3.5), Reapply Software Life Cycle (AP.1.9.3), Reengineering (AP.1.9.5)
Quality measures category	Reliability (QM.2.1.2), Availability (QM.2.1.1), Maintenance Control (QM.5.1.2.3), Productivity (QM.5.2)
Computing reviews category	not available

References and Information Sources

[Humphrey 95] Humphrey, Watts. *A Discipline for Software Engineering.* Reading, MA: Addison-Wesley Publishing Company, 1995.

[Humphrey 96a] Humphrey, Watts. "Using a Defined and Measured Personal Software Process." *IEEE Software 13*, 3 (May 1996): 77-88.

[Humphrey 96b] Humphrey, Watts. "The PSP and Personal Project Estimating." *American Programmer 9,* 6 (June 1996): 2-15.

[Paulk 95] Paulk, Mark C. *The Capability Maturity Model: Guidelines for Improving the Software Process.* Reading, MA: Addison-Wesley, 1995.

Author Dan Burton, SEI
dburton@sei.cmu.edu

External Reviewer(s) Archie Andrews, SEI
Watts Humphrey, SEI
Mark Paulk, SEI

Last Modified 10 Jan 97

Public Key Digital Signatures

ADVANCED

Note

We recommend Computer System Security— an Overview, pg. 129, as prerequisite reading for this technology description.

Purpose and Origin

Public key digital signature techniques provide data integrity and source authentication capabilities to enhance data trustworthiness in computer networks. This technology uses a combination of a message authentication code (MAC) to guarantee the integrity of data and unique features of paired public and secret keys associated with public key cryptography to uniquely authenticate the sender [Schneier 96, Abrams 95]. This technology was first defined in the early 1980s with the development of public key cryptography but has received renewed interest as an authentication mechanism on the Internet.

Technical Detail

Trustworthiness of data received by a computer from another computer is a function of the security capabilities of both computers and the communications between them. One of the fundamental objectives of computer security is data integrity [White 96]. Two aspects of data integrity are improved by public key digital signature techniques. These are sender authentication and data integrity verification. Positive authentication of the message source is provided by the unique relationship of the two encryption keys used in public key cryptography. Positive verification of message integrity is provided by the use of a message authentication code (sometimes called a manipulation detection code or a cryptographic checksum) that is produced by a message digest (sometimes called a data hashing) function. The use of a message authentication code and public key cryptography are combined in the public key digital signature techniques technology.

Sender authentication. Public key cryptography uses two paired keys. These are the public key and the secret key (sometimes called the private key), which are related to each other mathematically. The public key is distributed to anyone that needs to encrypt a message destined for the holder of the secret key. The secret key is not known to anyone but the holder of the secret key. Because of the mathematical relationship of the keys, data encrypted with the public key can only be decrypted with the secret key. Another feature of the paired key relationship is that if a message can be successfully decrypted with the public key then it must have been encrypted with the secret key. Therefore, any message decrypted by a holder of the public key must have been sent by the holder of the secret key. This is used to authenticate the source of a message. Public key cryptography can use one of several algorithms but the most common one is the Revisit, Shamir, and Adleman (RSA) algorithm. It is used

to produce the paired keys and to encrypt or decrypt data using the appropriate key.

Data integrity verification. Message digest functions produce a single large number called the message authentication code (MAC) that is *unique*[1] to the total combination and position of characters in the message being digested. The message digest function distributed with RSA is called the MD5 message digest function. It produces a *unique* 128 bit number for each different message digested. If even one character is changed in the message, a dramatically-different 128 bit number is generated.

The overall process for using Public Key Digital Signatures to verify data integrity is shown in Figure 22.

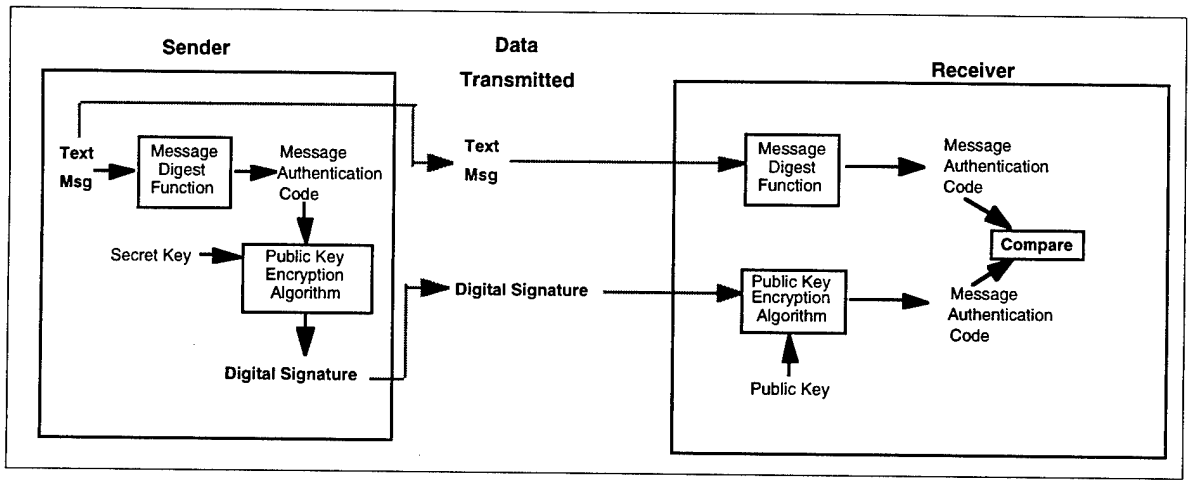


Figure 22: Public Key Digital Signatures

The Digital Signature of a message is produced in two steps:

1. The sender of the message uses the message digest function to produce a message authentication code (MAC).
2. This MAC is then encrypted using the private key and the public key encryption algorithm. This encrypted MAC is attached to the message as the digital signature.

The receiver of the message uses the public key to decrypt the digital signature. If it is decrypted successfully, the receiver of the message knows it came from the holder of the secret key. The receiver then uses the

1. Of course they are not absolutely unique. We say unique here because it is extremely unlikely statistically for two files to have the same MAC and, more importantly, it is extremely difficult for an attacker/malicious user to create/craft two files having the same MAC.

message digest function to calculate the MAC associated with the received message contents. If this number compares to the one decrypted from the Digital Signature, the message was received unaltered and data integrity is assured. This technique provides data source authentication and verification of message content integrity.

There are many message digest functions and public key encryption algorithms that may be used in developing the public key digital signature technique. A discussion of these alternative algorithms and their merits is in Schneier [Schneier 96].

Usage Considerations

This technology is most likely to be used in networks of computers where all the communication paths can not be physically protected and where the integrity of data and sender authenticity aspects of trustability are essential. Military C4I networks and banking networks that are on a widespread local area network or a wide area network are prime examples of this use.

Implementation of the public key digital signature techniques establishes additional requirements on a network. The same message digest functions and public key cryptography algorithm used to process the digital signature must be used by both the sender and receiver. Secret/public key pairs must be generated and maintained. Public keys must be distributed and secret keys protected.

Maturity

The components of this technology, public key encryption and message digest functions, have been in use since the early 1980s. The combined technology is mature and is available in implementations that range from small networks of PCs to protection of data being transferred over the Internet.

The algorithms supporting public key digital signatures have historically consumed large amounts of processing power. However, given recent advances in processors used in PCs and workstations; this is no longer a concern in most circumstances of use.

Costs and Limitations

Using this technology requires network management personnel with knowledge of public key cryptography and the use of software that implements public key and digital signature algorithms. It also requires security personnel that can generate, distribute, and control encryption/decryption keys and respond to the loss or compromise of keys.

Dependencies

Public key cryptography and message digest functions.

Alternatives

Data integrity and authentication can be provided by a combination of dedicated circuits, integrity protocols, and procedural control of sources

and destinations. These approaches are not foolproof and can be expensive. Data integrity and authentication can also be provided using private key encryption and a third party arbitrator. This approach has the disadvantage that a third party must be trusted and the data must be encrypted and decrypted twice with two separate private keys.

Index Categories

Name of technology	Public Key Digital Signatures
Application category	System Security (AP.2.4.3)
Quality measures category	Trustworthiness (QM.2.1.4)
Computing reviews category	Computer-Communication Networks Security and Protection (C.2.0), Security and Protection (K.6.5)

References and Information Sources

[Abrams 95] Abrams, Marshall D.; Jajodia, Sushil; & Podell, Harold J. *Information Security An Integrated Collection of Essays*. Los Alamitos, CA: IEEE Computer Society Press, 1995.

[Garfinkel 95] Garfinkel, Simpson. *PGP: Pretty Good Privacy*. Sebastopol, CA: O'Reilly & Associates, 1995.

[Russel 91] Russel, Deborah & Gangemi, G. T. Sr. *Computer Security Basics*. Sebastopol, CA: O'Reilly & Associates,1991.

[Schneier 96] Schneier, Bruce. *Applied Cryptography*. New York, NY: John Wiley & Sons, 1996.

[White 96] White, Gregory B.; Fisch, Eric A.; & Pooch, Udo W. *Computer System and Network Security*. Boca Raton, FL: CRC Press 1996.

Author

Tom Mills, Loral
TMILLS@ccs.lmco.com

External Reviewer(s)

Jim Ellis, SEI

Last Modified

10 Jan 97

Rate Monotonic Analysis

COMPLETE

Purpose and Origin

Rate Monotonic Analysis (RMA) is a collection of quantitative methods and algorithms that allows engineers to specify, understand, analyze, and predict the timing behavior of real-time software systems, thus improving their *dependability* and *evolvability*.

RMA grew out of the theory of fixed priority scheduling. A theoretical treatment of the problem of scheduling periodic tasks was first discussed by Serlin in 1972 [Serlin 72] and then more comprehensively treated by Liu and Layland in 1973 [Liu 73]. They studied an idealized situation in which all tasks are periodic, do not synchronize with one another, do not suspend themselves during execution, can be instantly preempted by higher priority tasks, and have deadlines at the end of their periods. The term "rate monotonic" originated as a name for the optimal task priority assignment in which higher priorities are accorded to tasks that execute at higher rates (that is, as a monotonic function of rate). Rate monotonic scheduling is a term used in reference to fixed priority task scheduling that uses a rate monotonic prioritization.

During the 1980s the limitations of the original theory were overcome and the theory was generalized to the point of being practicable for a large range of realistic situations encountered in the design and analysis of real-time systems [Sha 91a]. RMA can be used by real-time system designers, testers, maintainers, and troubleshooters, as it provides

- mechanisms for predicting real-time performance
- structuring guidelines to help ensure performance predictability
- insight for uncovering subtle performance problems in real-time systems

This body of theory and methods is also referred to as generalized rate monotonic scheduling (GRMS), a codification of which can be found in Klein [Klein 93].

Technical Detail

RMA provides the analytical foundation for understanding the timing behavior of real-time systems that must manage many concurrent threads of control. Real-time systems often have stringent latency requirements associated with each thread that are derived from the environmental processes with which the system is interacting. RMA provides the basis for

predicting whether such latency requirements can be satisfied. Some of the important factors that are used in RMA calculations include:

- the worst-case execution time of each thread of control
- the minimum amount of time between successive invocations of each thread
- the priority levels associated with the execution of each thread
- sources of overhead such as those due to an operating system
- delays due to interprocess communication and synchronization
- allocation of threads of control to physical resources such as CPUs, buses, and networks

These factors and other aspects of the system design are used to calculate worst-case latencies for each thread of control. These worst-case latencies are then compared to each thread's timing requirements to determine if the requirement can be satisfied.

A problem commonly revealed as a result of rate monotonic analysis is priority inversion. Priority inversion is a state in which the execution of a higher priority thread is forced to wait for a resource while a lower priority thread is using the resource. Not all priority inversion can be avoided but proper priority management can reduce priority inversion. For example, priority inheritance is a useful technique for reducing priority inversion in cases where threads must synchronize [Rajkumar 91].

Since RMA is an analytic approach that can be used before system integration to determine if latency requirements will be met, it can result in significant savings in both system resources and development time.

Usage Considerations

RMA is most suitable for systems dominated by a collection of periodic or sporadic processes (i.e., processes with minimum inter-arrival intervals), for which the processing times can be bounded and are without excessive variability. RMA is also primarily focused on hard deadlines rather than soft. However, soft deadlines can be handled through the use of server mechanisms that allocate time to tasks with soft deadlines in a manner that ensures that hard deadlines are still met. Still, with soft deadline tasks, the aperiodic server predictions work best when the workload is primarily periodic.

Systems in which worst-case executions are realized very infrequently or in which there is no minimum inter-arrival interval between thread invocations might *not* be suitable for RMA analysis. For example, consider multimedia applications where voice and data transmissions involve a great deal of variability. Principles of RMA, such as priority representation, priority arbitration, and priority inheritance, can be used in multime-

dia systems to reduce response times and meet deadlines at relatively high levels of use. However, deadlines in such environments may not be hard, and execution times can be stochastic, two requirements that are *not* currently handled well in the RMA framework. When most of the workload is aperiodic, one needs to move to queueing theory.

Maturity

Indicators of RMA maturity include the following:

- In 1989 IBM applied RMA to a sonar training system, allowing them to discover and correct performance problems [Lucas 92].
- Since 1990, RMA was recommended by IBM Federal Sector Division (now Lockheed Martin) for its real-time projects.
- RMA was successfully applied to active and passive sonar of a major submarine system of US Navy.
- RMA was selected by the European Space Agency as the baseline theory for its Hard Real-Time Operating System Project.
- The applicability of RMA to a typical avionics application was demonstrated [Locke 91].
- RMA was adopted in 1990 by NASA for development of real-time software for the space station data management subsystem. In 1992 Acting Deputy Administrator of NASA, Aaron Cohen stated, "Through the development of rate monotonic scheduling, we now have a system that will allow (Space Station) Freedom's computers to budget their time to choose [among] a variety of tasks, and decide not only which one to do first but how much time to spend in the process."
- Magnavox Electronics Systems Company incorporated RMA into real-time software development [Ignace 94].
- RMA principles have influenced the design and development of the following standards:
  - IEEE Futurebus+ [Sha 91b]
  - POSIX
  - Ada 95 (see pg. 67)
- Tool vendors provide the capability to analyze real-time designs using RMA. RMA algorithms, such as priority inheritance, have been used by operating system and Ada compiler vendors.

Costs and Limitations

Case studies of RMA adoption show that "While RMA does require engineers to re-frame their understanding of scheduling issues to a more abstract level, only moderate training is required for people to be effective in using the technology" [Fowler 93]. A short (1-2 day) tutorial is usually sufficient to gain a working knowledge of RMA.

Additionally, the studies found "RMA can be incorporated into software engineering processes with relative ease over a period of several months.... RMA can be adopted incrementally; its adoption can range

from application to an existing system by one engineer to application across an entire division as standard practice in designing new systems" [Fowler 93].

RMA can be applied with varying degrees of detail. Qualitative analysis through the application of design and trouble shooting heuristics can be very effective. Simple quantitative analysis using back-of-the-envelope calculations quickly yields insight into system timing behavior. More precise quantitative analysis can be performed as more precise system measurements become available during the development activity.

Dependencies

Application performance is influenced by system components such as operating systems networks and communication protocols. Therefore, it is important for such system components to be designed with RMA in mind.

Complementary Technologies

Simulation is often used to gain insight into a system's performance. Simulation can be used to corroborate RMA's performance predictions. Queueing theory is complementary to RMA. Whereas RMA is used to predict worst-case latencies when bounds can be placed on arrival dates and execution times, queueing theory can be used to predict average-case behavior when arrival rates and execution times are described stochastically. Together RMA and queuing theory solve a wide set of performance problems.

Index Categories

Name of technology	Rate Monotonic Analysis
Application category	Detailed Design (AP.1.3.5), System Analysis and Optimization (AP.1.3.6), Code (AP.1.4.2), Performance Testing (AP.1.5.3.5), Reapply Software Life Cycle (AP.1.9.3), Reengineering (AP.1.9.5)
Quality measures category	Real-Time Responsiveness/Latency (QM.2.2.2), Maintainability (QM.3.1), Reliability (QM.2.1.2)
Computing reviews category	Real-Time Systems (C.3)

References and Information Sources

[Audsley 95] Audsley, N.C., et al. "Fixed Priority Pre-Emptive Scheduling: An Historical Perspective." *Real Time Systems 8,* 2-3. (March-May 1995): 173-98.

[Fowler 93] Fowler, P. & Levine, L. *Technology Transition Push: A Case Study of Rate Monotonic Analysis* Part 1 (CMU/SEI-93-TR-29). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University,1993.

[Ignace 94] Ignace, S. J.; Sedlmeyer, R. L.; & Thuente, D. J. "Integrating Rate Monotonic Analysis into Real-Time Software Development," 257-274. *IFIP Transactions, Diffusion, Transfer and Implementation of Information Technology* (A-45). Pittsburgh, PA, October 11-13, 1993. The Netherlands: International Federation of Information Processing, 1994.

[Klein 93] Klein, M.H., et al. *A Practitioners' Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems.* Boston, MA: Kluwer Academic Publishers, 1993.

[Lehoczky 94] Lehoczky, J.P. "Real-Time Resource Management Techniques," 1011-1020. *Encyclopedia of Software Engineering.* New York, NY: J. Wiley & Sons, 1994.

[Liu 73] Liu, C. L. & Layland, J. W. "Scheduling Algorithms for Multi-Programming in a Hard Real-Time Environment." *Journal of the Association for Computing Machinery 20,* 1 (January 1973): 40-61.

[Locke 91] Locke, C.D.; Vogel, D.R.; & Mesler, T.J. "Building a Predictable Avionics Platform in Ada: a Case Study," 181-189. *Proceedings of the Twelfth Real-Time Systems Symposium.* San Antonio, TX, December 4-6, 1991. Los Alamitos, CA: IEEE Computer Society Press, 1991.

[Lucas 92] Lucas, L. & Page, B. "Tutorial on Rate Monotonic Analysis." *Ninth Annual Washington Ada Symposium.* McLean, VA, July 13-16, 1992. New York, NY: Association for Computing Machinery, 1992.

[Rajkumar 91] Rajkumar, Ragunathan. *Synchronization in Real-Time Systems: A Priority Inheritance Approach.* Boston, MA: Kluwer Academic Publishers, 1991.

[Serlin 72] Serlin, O. "Scheduling of Time Critical Processes," 925-932. *Proceedings of the Spring Joint Computer Conference.* Atlantic City, NJ, May 16-18, 1972. Montvale, NJ: American Federation of Information Processing Societies, 1972.

[Sha 91a] Sha, Klein & Goodenough, J. "Rate Monotonic Analysis for Real-Time Systems," 129-155. *Foundations of Real-Time Computing: Scheduling and Resource Management.* Boston, MA: Kluwer Academic Publishers, 1991.

[Sha 91b] Sha, L.; Rajkumar, R.; & Lehoczky, J. P. "Real-Time Computing with IEEE Futurebus+." *IEEE Micro 11,* 3 (June 1991): 30-38.

Author Mark Klein, SEI
mk@sei.cmu.edu

External Reviewer(s)	Mike Gagliardi, SEI John Goodenough, SEI John Lehoczky, Professor, Statistics Department, Carnegie Mellon University Ray Obenza, SEI Raj Rajkumar, Carnegie Mellon University Lui Sha, SEI
Last Modified	10 Jan 97

Reference Models, Architectures, Implementations— An Overview

ADVANCED

Purpose and Origin

Much confusion exists regarding the definition, applicability, and scope of the terms *reference model*, *architecture*, and *implementation*. Understanding these terms facilitates understanding legacy system designs and how to migrate them to more open systems. The purpose of this technology description is to provide definitions, and more importantly, to describe how the terms are related.

Technical Detail

Reference model. A reference model is a description of all of the possible software components, component services (functions), and the relationships between them (how these components are put together and how they will interact). Examples of commonly-known reference models include the following:

- the Technical Architecture for Information Management (TAFIM) reference model (see pg. 361)
- the Reference Model for Frameworks of Software Engineering Environments [ECMA 93]
- Project Support Environment Reference Model (PSERM)
- the Tri-Service Working Group Open Systems Reference Model

Architecture. An architecture is a description of a subset of the reference model's component services that have been selected to meet a specific system's requirements. In other words, not all of the reference model's component services need to be included in a specific architecture. There can be many architectures derived from the same reference model. The associated standards and guidelines for each service included in the architecture form the open systems architecture and become the criteria for implementing the system.

Implementation. The implementation is a product that results from selecting (e.g., commercial-off-the-shelf), reusing, building and integrating software components and component services according to the specified architecture. The selected, reused, and/or built components and component services must comply 100% with the associated standards and guidelines for the implementation to be considered compliant.

Usage Considerations

Figure 23 attempts to show the interrelationships of these concepts using the TAFIM as an example. TAFIM provides the reference model and a number of specific architectures can be derived from the TAFIM reference model based on specific program requirements. From there a number of implementations may be developed based on the products

selected to meet the architecture's services, so long as these products meet the required standards and guidelines. For instance, in one implementation, the product ORACLE might be selected and used to meet some of the data management services. In another implementation, the product Sybase might be selected and used.

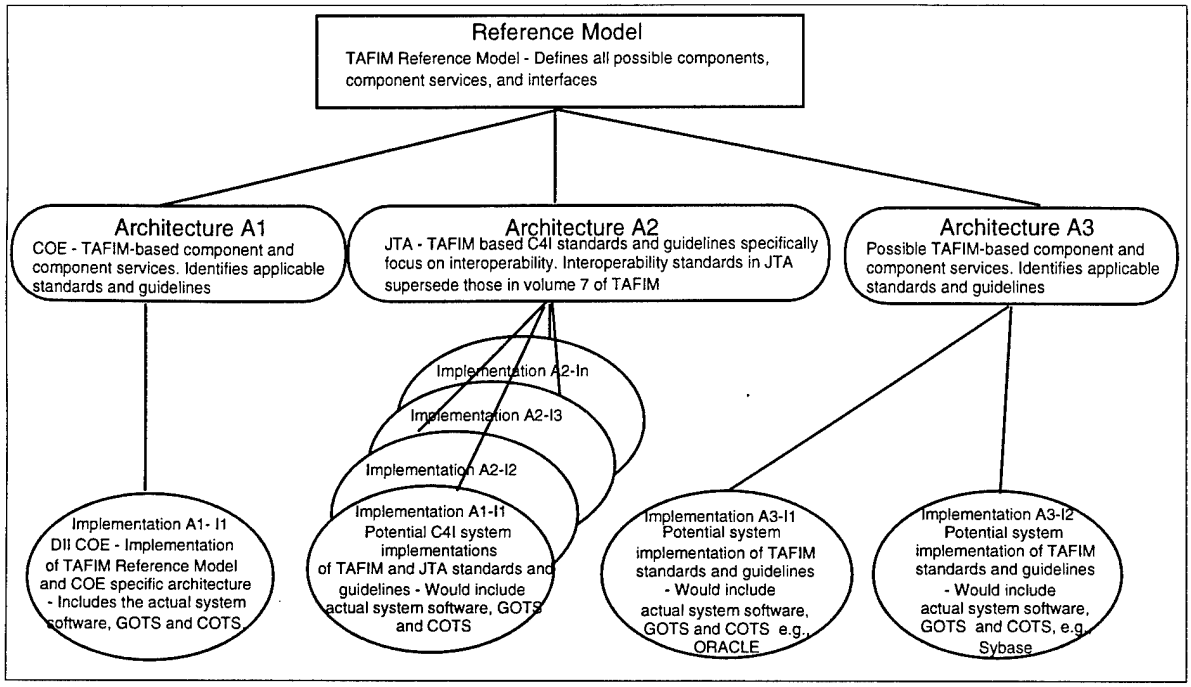


Figure 23: Reference Model, Architecture, and Implementation

Index Categories

Name of technology	Reference Models, Architectures, Implementations— An Overview
Application category	Software Architecture Models (AP.2.1.1), Software Architecture (AP.2.1)
Quality measures category	Maintainability (QM.3.1), Interoperability (QM.4.1), Portability (QM.4.2)
Computing reviews category	Distributed Systems (C.2.4), Software Engineering Design (D.2.10)

References and Information Sources

[ECMA 93] *Reference Model for Frameworks of Software Engineering Environments*, 3rd Edition (NIST Special Publication 500-211/Technical Report ECMA TR/55). Prepared jointly by NIST and the European Computer

Manufacturers Association (ECMA). Washington, DC: U.S. Government Printing Office, 1993.

[Myers 96] Meyers, Craig & Oberndorf, Tricia. *Open Systems: The Promises and the Pitfalls.* Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1996.

Author Darleen Sadoski, GTE
sadoski.darleen@mail.ndhm.gtegsc.com

External Reviewer(s) Tricia Oberndorf, SEI

Last Modified 10 Jan 97

Remote Procedure Call — ADVANCED

Note

We recommend Middleware, pg. 251, as prerequisite reading for this technology description.

Purpose and Origin

Remote Procedure Call (RPC) is a client/server infrastructure that increases the *interoperability*, *portability*, and *flexibility* of an application by allowing the application to be distributed over multiple heterogeneous platforms. It reduces the *complexity* of developing applications that span multiple operating systems and network protocols by insulating the application developer from the details of the various operating system and network interfaces— function calls are the programmer's interface when using RPC [Rao 1995].

The concept of RPC has been discussed in literature as far back as 1976, with full-scale implementations appearing in the late 1970s and early 1980s [Birrell 84].

Technical Detail

In order to access the remote server portion of an application, special function calls, RPCs, are embedded within the client portion of the client/server application program. Because they are embedded, RPCs do not stand alone as a discreet middleware layer. When the client program is compiled, the compiler creates a local stub for the client portion and another stub for the server portion of the application. These stubs are invoked when the application requires a remote function and typically support synchronous calls between clients and servers. These relationships are shown in Figure 24 [Steinke 95].

By using RPC, the complexity involved in the development of distributed processing is reduced by keeping the semantics of a remote call the same whether or not the client and server are collocated on the same system. However, RPC increases the involvement of an application developer with the complexity of the master-slave nature of the client/server mechanism.

RPC increases the flexibility of an architecture by allowing a client component of an application to employ a function call to access a server on a remote system. RPC allows the remote component to be accessed without knowledge of the network address or any other lower-level information. Most RPCs use a synchronous, request-reply (sometimes referred to as "call/wait") protocol which involves blocking of the client until the server fulfills its request. Asynchronous ("call/nowait") implementations are available but are currently the exception.

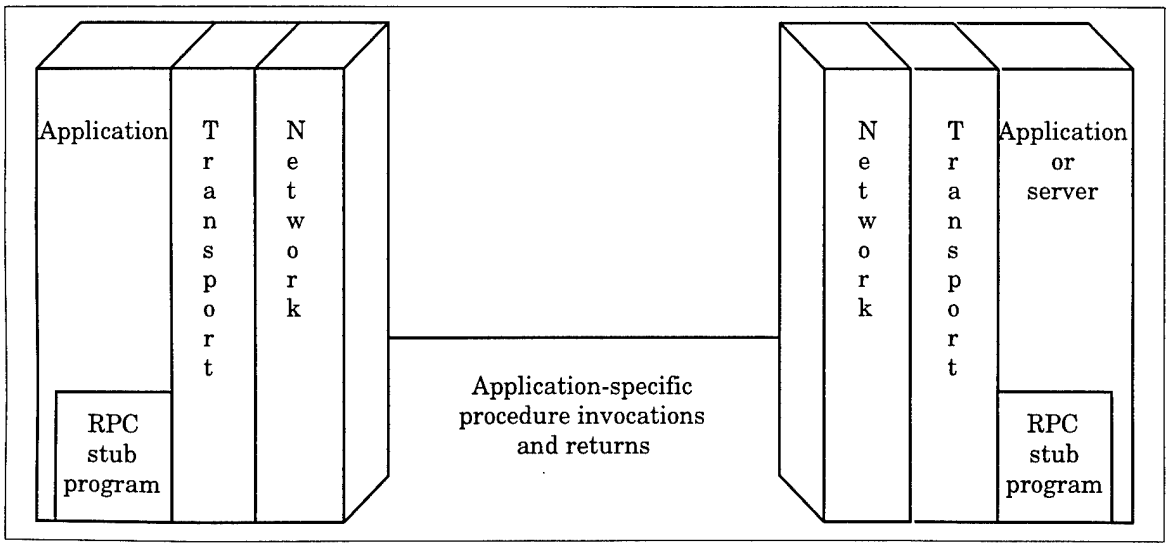


Figure 24: Remote Procedure Calls

RPC is typically implemented in one of two ways:

1. within a broader, more encompassing propriety product
2. by a programmer using a proprietary tool to create client/server RPC stubs

Usage Considerations

RPC is appropriate for client/server applications in which the client can issue a request and wait for the server's response before continuing its own processing. Because most RPC implementations do not support peer-to-peer, or asynchronous, client/server interaction, RPC is not well-suited for applications involving distributed objects or object-oriented programming (see pg. 287).

Asynchronous and synchronous mechanisms each have strengths and weaknesses that should be considered when designing any specific application. In contrast to asynchronous mechanisms employed by message-oriented middleware (see pg. 247), the use of a synchronous request-reply mechanism in RPC requires that the client and server are always available and functioning (i.e., the client or server is not blocked). In order to allow a client/server application to recover from a blocked condition, an implementation of a RPC is required to provide mechanisms such as error messages, request timers, retransmissions, or redirection to an alternate server. The complexity of the application using a RPC is dependent on the sophistication of the specific RPC implementation (i.e., the more sophisticated the recovery mechanisms supported by RPC, the less complex the application utilizing the RPC is required to be). RPCs that implement asynchronous mechanisms are very few and are difficult (complex) to implement [Rao 1995].

When utilizing RPC over a distributed network, the performance (or load) of the network should be considered. One of the strengths of RPC is that the synchronous, blocking mechanism of RPC guards against overloading a network, unlike the asynchronous mechanism of message-oriented middleware (MOM) (see pg. 247). However, when recovery mechanisms, such as retransmissions, are employed by an RPC application, the resulting load on a network may increase, making the application inappropriate for a congested network. Also, because RPC uses static routing tables established at compile-time, the ability to perform load balancing across a network is difficult and should be considered when designing an RPC-based application.

Maturity

Tools are available for a programmer to use in developing RPC applications over a wide variety of platforms, including Windows (3.1, NT, 95), Macintosh, 26 variants of UNIX, OS/2, NetWare, and VMS [Steinke 1995]. RPC infrastructures are implemented within the Distributed Computing Environment (DCE) (see pg. 167), and within Open Network Computing (ONC), developed by Sunsoft, Inc. These two RPC implementations dominate the current middleware (see pg. 251) market [Rao 1995].

Costs and Limitations

RPC implementations are nominally incompatible with other RPC implementations, although some are compatible. Using a single implementation of a RPC in a system will most likely result in a dependence on the RPC vendor for maintenance support and future enhancements. This could have a highly negative impact on a system's flexibility, maintainability, portability, and interoperability.

Because there is no single standard for implementing an RPC, different features may be offered by individual RPC implementations. Features that may affect the design and cost of a RPC-based application include the following:

- support of synchronous and/or asynchronous processing
- support of different networking protocols
- support for different file systems
- whether the RPC mechanism can be obtained individually, or only bundled with a server operating system

Because of the complexity of the synchronous mechanism of RPC and the proprietary and unique nature of RPC implementations, training is essential even for the experienced programmer.

Alternatives

Other middleware technologies that allow the distribution of processing across multiple processors and platforms are

- Object Request Brokers (ORB) (see pg. 291)
- Distributed Computing Environment (DCE) (see pg. 167)
- Message-oriented Middleware (MOM) (see pg. 247)
- OLE/COM (see pg. 271)
- Transaction Processing Monitor Technology (see pg. 373)
- Three Tier Architectures (see pg. 367)

Complementary Technologies

RPC can be effectively combined with message-oriented middleware (MOM) technology (see pg. 247)— MOM can be used for asynchronous processing.

Index Categories

Name of technology	Remote Procedure Call
Application category	Client/Server (AP.2.1.2.1), Client/Server Communication (AP.2.2.1)
Quality measures category	Maintainability (QM.3.1), Interoperability (QM.4.1), Portability (QM.4.2), Complexity (QM.3.2.1)
Computing reviews category	Distributed Systems (C.2.4)

References and Information Sources

[Birrell 84] Birrell, A.D. & Nelson, B.J. "Implementing Remote Procedure Calls." *ACM Transactions on Computer Systems 2,* 1 (February 1984): 39-59.

[Rao 95] Rao, B.R. "Making the Most of Middleware." *Data Communications International 24*, 12 (Sept. 1995): 89-96.

[Steinke 95] Steinke, Steve. "Middleware Meets the Network." *LAN Magazine* (December 1995): 56-63.

[Thekkath 93] Thekkath, C.A. & Levy, H.M. "Limits to Low-Latency Communication on High-Speed Networks." *ACM Transactions on Computer Systems 11*, 2 (May 1993): 179-203.

Author

Cory Vondrak, TRW, Redondo Beach, CA

Last Modified

10 Jan 97

Requirements Tracing

ADVANCED

Purpose and Origin

The development and use of requirements tracing techniques originated in the early 1970s to influence the *completeness*, *consistency*, and *traceability* of the requirements of a system. They provide an answer to the following questions:

- What mission need is addressed by a requirement?
- Why is this requirement here?
- Where is a requirement implemented?
- Is this requirement necessary?
- How do I interpret this requirement?
- What design decisions affect the implementation of a requirement?
- Are all requirements allocated?
- Why is the design implemented this way and what were the other alternatives?
- Is this design element necessary?
- Is the implementation compliant with the requirements?
- What acceptance test will be used to verify a requirement?
- Are we done?
- What is the impact of changing a requirement [SPS 94]?

The purpose of this technology description is to introduce the key concepts of requirements tracing. Detailed discussions of the individual technologies can be found in the referenced technology descriptions.

Technical Detail

Requirements traceability is defined as the ability to describe and follow the life of a requirement, in both a forward and backward direction (i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through periods of ongoing refinement and iteration in any of these phases) [Gotel 95]. It can be achieved by using one or more of the following techniques:

- Cross referencing, which involves embedding phrases like "see section x" throughout the project documentation (e.g., tagging, numbering, or indexing of requirements, and specialized tables or matrices that track the cross references).
- Using specialized templates and integration or transformation documents to store links between documents created in different phases of development.
- Restructuring the documentation in terms of an underlying network or graph to keep track of requirements changes (e.g., assumption-

based truth maintenance networks, chaining mechanisms, constraint networks, and propagation) [Gotel 95].

Usage Considerations

For any given project, a key milestone (or step) is to determine and agree upon requirements traceability details. Initially, three important questions need to be answered before embarking on any particular requirements traceability approach:

1. What needs to be traceable?
2. What linkages need to be made?
3. How, when, and who should establish and maintain the resulting database?

Once the questions are answered, then selection of an approach can be made. One approach could be the structured use of general-purpose tools (e.g., hypertext editors, word processors, and spreadsheets) configured to support cross-referencing between documents. For large software development projects, an alternative approach could be the use of a dedicated workbench centered around a database management system providing tools for documenting, parsing, editing, decomposing, grouping, linking, organizing, partitioning, and managing requirements. The following table describes the strengths and weaknesses of each of the approaches.

Approaches	Strengths	Weaknesses
General purpose tools	• readily available • flexible • good for small projects	• need to be configured to support Requirements Traceability (RT) • potential high RT maintenance cost • limited control over RT information • potential limited integration with other software development tools
Workbenches	• fine-grained forward, backward, horizontal, and vertical RT • RT results may facilitate later development activities (i.e., testing) • suitable for large projects	• depend upon stakeholder buy-in • manual intervention may be required • RT in later development phases may be difficult

Regardless of the approach taken, requirements tracing requires a combination of models (i.e., representation forms), methods (i.e., step by step processes), and/or languages (i.e., semiformal and formal) that incorporate the above techniques. Some examples of requirements tracing methods are discussed in the following technology descriptions:

- Feature-Based Design Rationale Capture Method for Requirements Tracing (pg. 181),
- Argument-Based Design Rationale Capture Methods for Requirements Tracing (pg. 91)

Maturity

Every major office tool manufacturer has spreadsheet and/or database capabilities that can be configured to support requirements tracing. There are at least ten commercial products that fall in the workbench category and support some level of requirements traceability. At a minimum, they provide bidirectional requirement linking to system elements; capture of allocation rationale, accountability, and test/validation; identification of inconsistencies; capabilities to view/trace links; verification of requirements; and a history of requirements changes. Environments to support requirements traceability past the requirements engineering phase of the system/software life cycle are being researched. Areas include the development of a common language, method, model, and database repository structure, as well as mechanisms to provide data exchange between different tools in the environment. Prototypes exist and at least one commercial product provides support for data exchange through its object-oriented database facilities.

Costs and Limitations

In general, the implementation of requirements tracing techniques within an organization should facilitate reuse and maintainability of the system. However, additional resources (time and manpower) to initially implement traceability processes (i.e., definition of traceability information, selection of automated tools, training, etc.) will be required. One case study found that the cost was more than twice the normal documentation cost associated with the development of a system of similar size and complexity. However, this was determined to be a one-time cost and the overall costs to maintain the software system are expected to be reduced. Almost immediate return was observed in the reduced amount of time to perform hardware upgrades [Ramesh 95].

Index Categories

Name of technology	Requirements Tracing
Application category	Requirements Tracing (AP.1.2.3)
Quality measures category	Completeness (QM.1.3.1), Consistency (QM.1.3.2), Traceability (QM.1.3.3), Effectiveness (QM.1.1), Reusability (QM.4.4), Understandability (QM.3.2), Maintainability (QM.3.1)
Computing reviews category	Software Engineering Tools and Techniques (D.2.2), Software Engineering Requirements/ Specifications (D.2.1)

References and Information Sources

[Bailin 90] Bailin, S., et al. "KAPTUR: Knowledge Acquisition for Preservation of Tradeoffs and Underlying Rationales," 95-104. *Proceedings of the 5th Annual Knowledge-Based Software Assistant Conference.* Liverpool, NY, September 24-28, 1990. Rome, NY: Rome Air Development Center, 1990.

[Gotel 95] Gotel, Orlena. *Contribution Structures for Requirements Traceability.* Doctoral Dissertation. London, England: Department of Computing, Imperial College, 1995.

[Ramesh 92] Ramesh, Balasubramaniam and Dhar, Vasant. "Supporting Systems Development by Capturing Deliberations During Requirements Engineering." *IEEE Transactions on Software Engineering 18,* 6 (June 1992): 498-510.

[Ramesh 95] Ramesh, B., et al. "Lessons Learned from Implementing Requirements Traceability." *Crosstalk, Journal of Defense Software Engineering 8,* 4 (April 1995): 11-15.

[Shum 94] Shum, Buckingham Simon & Hammond, Nick. "Argumentation-Based Design Rationale: What Use at What Cost?" *International Journal of Human-Computer Studies 40,* 4 (April 1994): 603-652.

[SPS 94] *Analysis of Automated Requirements Management Capabilities.* Melbourne, FL: Software Productivity Solutions, 1994.

Author Liz Kean, Rome Laboratory
liz@se.rl.af.mil

External Reviewer(s) Brian Gallagher, SEI

Last Modified 10 Jan 97

Rule-Based Intrusion Detection ADVANCED

Note

We recommend Intrusion Detection, pg. 217, as prerequisite reading for this technology description.

Purpose and Origin

Due to the voluminous, detailed nature of system audit data — some of which may have little if any meaning to a human reviewer — and the difficulty of discriminating between normal and intrusive behavior, one approach taken by developers of intrusion detection systems is to use expert systems technology to analyze automatically audit trail data for intrusion attempts [Lunt 93]. These security systems, known as rule-based intrusion detection (RBID) systems, can be used to analyze system audit trails for pending or completed computer security violations. This emerging technology seeks to increase the *availability* of computer systems by automating the detection and elimination of intrusions.

Technical Detail

Rule-based intrusion detection (RBID) is predicated on the assumption that intrusion attempts can be characterized by sequences of user activities that lead to compromised system states. RBID systems are characterized by their expert system properties that fire rules[1] when audit records or system status information begin to indicate illegal activity [Ilgun 93]. These predefined rules typically look for high-level state change patterns observed in the audit data compared to predefined penetration state change scenarios. If an RBID expert system infers that a penetration is in process or has occurred, it will alert the computer system security officers and provide them with both a justification for the alert and the user identification of the suspected intruder.

---

1. In an expert system, knowledge about a problem domain is represented by a set of rules. These rules consist of two parts:

   1) The antecedent, which defines when the rule should be applied. An expert system will use pattern matching techniques to determine when the observed data matches or satisfies the antecedent of a rule.

   2) The consequent, which defines the action(s) that should be taken if its antecedent is satisfied.

   A rule is said to be "fired" when the action(s) defined in its consequent are executed. For RBID systems, rule antecedents will typically be defined in terms of audit trail data, while rule consequents may be used to increase or decrease the level of monitoring of various entities, or they may be used to notify system administration personnel about significant changes in system state.

There are two major approaches to rule-based intrusion detection:

1. *State-based.* In this approach, the rule base is codified using the terminology found in the audit trails. Intrusion attempts are defined as sequences of system state— as defined by audit trail information— leading from an initial, limited access state to a final compromised state [Ilgun 93].
2. *Model-based.* In this approach, known intrusion attempts are modeled as sequences of user behavior; these behaviors may then be modeled, for example, as events in an audit trail. Note, however, that the intrusion detection system itself is responsible for determining how an identified user behavior may manifest itself in an audit trail. This approach has many benefits, including the following:
   - More data can be processed, because the technology allows you to narrow the focus of the data selectively.
   - More intuitive explanations of intrusion attempts are possible.
   - The system can predict the intruder's next action.

Usage Considerations

RBID rule bases are affected by system hardware or software changes and require updates by system experts as the system is enhanced or maintained. The protection afforded by RBID systems would be most useful in an environment where physical protection of the computer system is not always possible (e.g., a battlefield situation), yet the data is of high value and requires stringent protection.

Maturity

Although RBID systems are in the research and early prototype stage, articles describing RBID systems date to at least the 1986 description of the Discovery system [Tener 86]. In 1987, Denning described an early, abstract model of a rule-based intrusion detection system (IDS) [Denning 87]; in 1989, Vaccarro and Liepins described the Wisdom and Sense system [Vaccarro 89]. More recent systems include USTAT [Ilgun 93] and the Intrusion Detection Expert System (IDES) [Lunt 93]; IDES combines statistical-based (see pg. 357) and model-based intrusion detection approaches to achieve a level of intrusion detection not feasible with either approach alone. Mukherjee describes several other recent RBID systems [Mukherjee 94]. Feasibility for an operational system has not yet been demonstrated.

Costs and Limitations

The use of RBID systems requires the following:

- personnel knowledgeable in rule-based systems, especially with respect to rule representation
- personnel who know how various activities may be represented in audit trails
- personnel experienced in intrusion detection and who have in-depth knowledge of the audit collection mechanism [Ilgun 93]

In addition to the costs associated with maintaining intrusion detection knowledge bases, there are several risks and limitations associated with this technology:

- Only known vulnerabilities and attacks are codified in the knowledge base. The knowledge base of rules is thus always playing "catch-up" with the intruders [Lunt 93].
- The representation of intrusion scenarios— especially with respect to state-based approaches— is not intuitive.

For these reasons, RBIDs cannot detect all intrusion attempts.

Like all intrusion detection systems, RBIDs will negatively affect system performance due to their collecting and processing of audit trail information. For example, early prototyping of a real-time RBID system on a UNIX workstation showed the algorithm was using up to 50% of the available processor throughput to process and analyze the audit trail [Ilgun 93].

Dependencies

Expert systems are an enabler for this technology.

Alternatives

Other automated approaches to intrusion detection include statistical-based approaches (see pg. 357) and approaches based on genetic algorithms. Manual examination of recorded audit data and online monitoring of access activity by knowledgeable system security personnel are the only other known alternatives.

Complementary Technologies

RBID systems can be used in conjunction with statistical-based intrusion detection systems (see pg. 357) to catch a wider variety of intrusion attempts, and authentication systems can be used to verify user identity.

Index Categories

Name of technology	Rule-Based Intrusion Detection
Application category	System Security (AP.2.4.3)
Quality measures category	Security (QM.2.1.5)
Computing reviews category	Operating Systems Security and Protection (D.4.6), Computer-Communication Networks Security and Protection (C.2.0), Security and Protection (K.6.5)

References and Information Sources

[Bell 76] Bell, D. E. & LaPadula, L. J. *Secure Computer System: Unified Exposition and Multics Interpretation* Rev. 1 (MTR-2997). Bedford, MA: MITRE Corporation, 1976.

[Denning 87] Denning, Dorothy E., et al. "Views for Multilevel Database Security." *IEEE Transactions on Software Engineering SE-13,* 2 (February 1987): 129-140.

[CSC 83] Computer Security Center. *Department of Defense Trusted Computer System Evaluation Criteria.* Fort George G. Meade, MD: DoD Computer Security Center, 1983.

[Ilgun 93] Ilgun, Koral. "USTAT: A Real-time Intrusion Detection System for UNIX," 16-28. *Proceedings of the 1993 Computer Society Symposium on Research in Security and Privacy.* Oakland, California, May 24-26, 1993. Los Alamitos, CA: IEEE Computer Society Press, 1993.

[Kemmerer 94] Kemmerer, Richard A. "Computer Security," 1153-1164. *Encyclopedia of Software Engineering.* New York, NY: John Wiley and Sons, 1994.

[Lunt 93] Lunt, Teresa F. "A Survey of Intrusion Detection Techniques." *Computers and Security 12,* 4 (June 1993): 405-418.

[Mukherjee 94] Mukherjee, Biswanath; Heberlein, L. Todd; & Levitt, Karl N. "Network Intrusion Detection." *IEEE Network 8,* 3 (May-June 1994): 26-41.

[Sundaram 96] Sundaram, Aurobindo. *An Introduction to Intrusion Detection* [online]. Available WWW <URL: http://www.acm.org/crossroads/xrds2-4.html> (1996).

[Tener 86] Tener, W. T. "Discovery: An Expert System in the Commercial Data Security Environment." *Computer Security Journal 6,* 1 (Summer 1990): 45.

[Vaccarro 89] Vaccarro, H. S. & Liepins, G. E. "Detection of Anomalous Computer Session Activity," 208-209. *Proceedings of the IEEE Symposium on Re-*

search in Security and Privacy. Oakland, California, May 1-3, 1989. Washington, DC: IEEE Computer Society Press, 1989.

[Ware 79] Ware, W. H. *Security Controls for Computer Systems.* Santa Monica, CA: The Rand Corporation, 1979.

Author Mark Gerken, Rome Laboratory
gerken@ai.rl.af.mil

Last Modified 10 Jan 97

Simple Network Management Protocol ADVANCED

Purpose and Origin

Simple network management protocol (SNMP) is a network management specification developed by the Internet Engineering Task Force (IETF), a subsidiary group of the IAB (Internet Activities Board) in the late 1980s in order to provide standard, *simplified*, and *extensible* management of LAN-based internetworking products such as bridges, routers, and wiring concentrators. [IETF 96, Henderson 1995]. SNMP was designed to reduce the *complexity* of network management and minimize the amount of resources required to accommodate it. SNMP provides for centralized, robust, *interoperable* management, along with the *flexibility* to allow for the management of vendor-specific variables.

Technical Detail

The term network management generally includes:[1]

- network monitoring
- control
- capacity planning
- troubleshooting

SNMP is best suited for network monitoring and capacity planning. SNMP does not provide even the basic troubleshooting information that can be obtained from simple network troubleshooting tools.

SNMPv1 is a simple request/response application-layer protocol which typically uses the User Datagram Protocol(UDP) for data delivery. It is designed to exchange management information between network managers and network managed systems. A network managed system can be any type of node residing on a network, such as computers, printers, and routers. Agents are software modules that run in a network managed system. An agent has access to information about the managed system. SNMP is part of the Internet network management architecture that contains: a SNMP manager, SNMP agent(s), and a management information base (MIB), which is basically a database of managed objects that resides on the SNMP agent.

1. This tends to be an SNMP-centric view: An alternative view of network management is defined by ISO, as part of X.700/FCAPS. That view of network management comprehends the following areas: fault management, configuration management, accounting management, performance management, and security management [X.700 96].

Attributes of managed objects may be monitored or set using the following operations:

- get an object instance from the agent
- get the next object instance from a table or list from an agents
- set object instances within an agent
- send events (traps) asynchronously to managers

The management application or user can define the relationship between the manager and the agent. This architecture is shown in Figure 25.

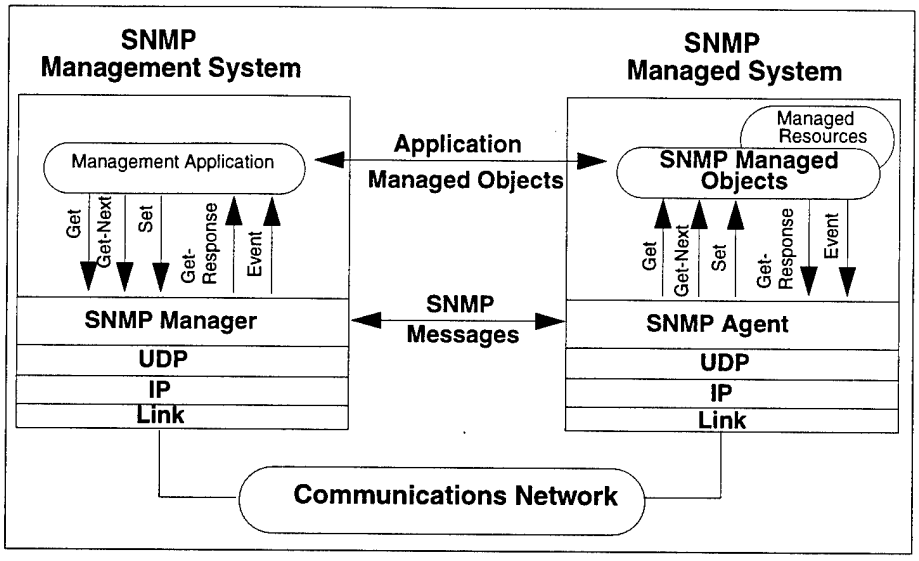


Figure 25: The SNMPv1 Architecture [Lake 96]

By specifying the protocol to be used between the manager and agent, SNMP allows products from different vendors (and their associated agents) to be managed by the same SNMP manager. A "proxy function" is also specified by SNMP to enable communication with non-SNMP devices to accommodate legacy equipment.

Two of SNMP's main attributes are that [Moorhead 95]

- It is simple to implement, making it easy for a vendor to accommodate it into its device.
- It does not call for large computational or memory resources from the devices that do accommodate it.

Network management, as defined by SNMP, is based on two types of activities, polling and interrupts. The SNMP manager polls for information gathered by each of the agents. Each agent has the responsibility of collecting information (e.g., performance statistics) pertaining to the device

it resides within and storing that information in the agent's own management information base (MIB). This information is sent to the SNMP manager in response to the manager's polling.

SNMP interrupts are driven by trap messages generated as a result of certain device parameters. These parameters can be either generic or specific to the device vendor information. Enterprise-specific trap messages are vendor proprietary and generally provide more device-specific detail.

Usage Considerations

Neither version of SNMP (SNMPv2 is described in the Maturity section, pg. 339) does an effective job at helping network managers isolate problem devices in large, complex networks. It sometimes becomes difficult for an SNMP manager to determine which network events/alarms are significant— all are treated equally. SNMPv1 provides information only on individual devices, not on how the devices work as a system.

The performance impact on the network being managed should be considered when using the polling scheme that SNMP uses for collecting information from distributed agents. A higher frequency of polling, which may be required to manage a network more effectively, will increase the overhead on a network, possibly resulting in a need for additional networking or processor resources. The frequency of polling can be controlled by the SNMP manager, but can be dependent on what kind of messages (generic or enterprise-specific) a device vendor supports. Many vendors offer generic trap messages on their devices rather than enterprise-specific messages, because it is easier and takes less time for the vendor to implement. Devices that provide only generic trap information must be polled frequently in order to obtain the granularity of information to manage the device effectively.

Maturity

SNMPv1 has been incorporated into many products and management platforms. It has been deployed by virtually all internetworking vendors—40 as of March 1995. It has been widely adopted for the enterprise networks and may be the manager of choice for the internetworking arena in the future because it is well-suited for managing TCP/IP networks. Yet it does have limitations, as discussed in Costs and Limitations, pg. 340.

The SNMPv2 (SNMP Version 2) specification included the following new capabilities:

- commands to support the coexistence of multiple/distributed managers and mid-level managers, increasing the flexibility and *scalability* of the network being managed
- enhanced *security* (known as "Secure SNMP") by specifying three layers of security
  - encryption based on the Data Encryption Standard
  - authentication
  - authorization
- improved efficiency of polling over SNMPv1 by allowing bulk transfers of data. This means that in some cases, using SNMPv2 instead of SNMPv1, network management can be provided over low-bandwidth, wide-area links.
- support for additional network protocols besides UDP/IP, for example, OSI, NetWare IPX/SPX and Appletalk [Broadhead 95]

However, SNMPv2 had not reached draft standard status within the IETF and was supported by few vendors as of May 1995— SNMPv2 has many unresolved issues. It may take years before SNMPv2 is widely accepted.

Costs and Limitations

The attractiveness of SNMP is its simplicity and associated relative ease of implementation. With this comes a price: e.g., the more fine grained things you want, the less likely it is that they will be available.

SNMP uses the underlying User Datagram Protocol (UDP) for data delivery, which does not ensure reliability of data transfer. The loss of data may or may not be a limitation to a network manager, depending on the criticality of the information being gathered and the frequency at which the polling is being performed.

SNMPv1 has minimal security capability. Because SNMPv1 lacks the control of unauthorized access to critical network devices and systems, it may be necessary to restrict the use of SNMP management to non-critical networks. Lack of authentication in SNMPv1 has led many vendors to not include certain commands, thus reducing extensibility and consistency across managed devices. SNMPv2 addresses these security problems but is messy and time-consuming to set up and administer (e.g., each MIB must be locally set up).

SNMP by itself does not provide any application programs or user interfaces in terms of plots, visual displays, and the like. These types of applications would have to be developed separately.

SNMP out-of-the-box can not be used to track information that is contained in application/user level protocols (e.g., radar track message, http, mail). However these might be accomplished through the use of a extensible (customized) SNMP agent that has user defined MIB.[1] It is important to note that a specialized or extensible network manager may be required for use with the customized agents.

There are also concerns about the use of SNMP in the real-time domain where bounded response, deadlines, and priorities are required.

SNMPv2 is intended to be able to coexist with existing SNMPv1, but in order to use SNMPv2 as the SNMP manager or to migrate from SNMPv1 to SNMPv2, all SNMPv1 compliant agents must be entirely replaced with SNMPv2 compliant agents— gateways or bilingual managers and proxy agents were not available to support the gradual migration as of early-1995.

Alternatives

Common management information protocol (CMIP) may be a better alternative for large, more complex networks or security-critical networks.

CMIP design is similar to SMNP and was developed to make up for SNMP's shortcomings. However, CMIP takes significantly more system resources than SNMP, is difficult to program, and is designed to run on the ISO protocol stack. (However, the technology standard used today in most systems is TCP/IP.)

The biggest feature in CMIP is that tasks can be performed or events can be triggered based upon the value of a variable or a specific condition. For example, when a computer can not reach its network fileserver for a predetermined number of times, an event can be generated to notify the appropriate personnel. With SNMP, this task would have to be performed by a user keeping track of failed attempts.

1. There is an MIB being developed for http [MIB 96], and the MIB for mail monitoring is now a proposed standard.

Index Categories

Name of technology	Simple Network Management Protocol
Application category	Protocols (AP.2.2.3), Network Management (AP.2.2.2)
Quality measures category	Maintainability (QM.3.1), Simplicity (QM.3.2.2), Complexity (QM.3.2.1), Efficiency/ Resource Utilization (QM.2.2), Scalability (QM.4.3), Security(QM.2.1.5)
Computing reviews category	Network Operations (C.2.3), Distributed Systems (C.2.4)

References and Information Sources

[Broadhead 95] Broadhead, Steve. "SNMP Too Simple for Security?" *Secure Computing* (April 1995): 24-29.

[Comer 91] Comer, Douglas. "Internetworking with TCP/IP." Englewood Cliffs, NJ: Simon and Schuster, 1991.

[Feit 94] Feit, Sidnie. *A Guide to Network Management.* New York, NY: McGraw Hill, 1994.

[Henderson 95] Henderson and Erwin. "SNMP Version 2: Not So Simple." *Business Communications Review 25,* 5 (May 1995): 44-48.

[Herman 94] Herman, James. "Network Computing Inches Forward." *Business Communications Review 24,* 5 (May 1994): 45-50.

[IETF 96] Internet Engineering Task Force home page [online]. Available WWW <URL: http://www.ietf.crni.reston.va.us/> (1996).

[Kapoor 94] Kapoor, K. "SNMP Platforms: What's Real, What Isn't." *Data Communications International 23,* 12 (September 1994): 115-18.

[Lake 96] Lake, Craig. *Simple Network Management Protocol (SNMP)* [online]. Available WWW <URL: http://www.rpm.com/whitepaper/snmp.html> (1996).

[MIB 96] *Development of an MIB for http* [online]. Available WWW <URL: http://http-mib.onramp.net/bof/> (1996).

[Moorhead 95] Moorhead, R.J. & Amirthalingam, K. "SNMP— An Overview of its Merits and Demerits," 180-3. *Proceedings of the Twenty-Seventh Southeastern Symposium on System Theory.* Starkvill, MS, March 12-14, 1995. Los Alamitos, CA: IEEE Computer Society Press, 1995.

[Phifer 94] Phifer, L.A. "Tearing Down the Wall: Integrating ISO and Internet Management." *Journal of Network and Systems Management 2,* 3 (September 1994): pp. 317-22.

[Rose 94] Rose, Marshall T. *The Simple Book: An Introduction to Internet Management.* Englewood Cliffs, NJ: Prentice-Hall, 1994.

[SNMP 96] *Simple Network Management Protocol* [online]. Available WWW <URL: http://www.snmp.com> and <URL: http://www.snmp.com/snmppages.html> (1996).

[Stallings 93] Stallings, William. *SNMP, SNMPv2, and CMIP: The Practical Guide to Network Management Standards.* Reading, MA: Addison-Wesley, 1993.

[Vallillee 96] Vallillee, Tyler. *SNMP & CMIP: An Introduction To Network Management* [online]. Available WWW <URL: http://www.inforamp.net/~kjvallil/t/snmp.html> (1996).

[Wellens 96] Wellens, Chris & Auerbach, Karl. "Towards Useful Management" [online]. *The Quarterly Newsletter of SNMP Technology, Comment, and Events(sm) 4,* 3 (July 1996). Available WWW <URL: http://simple-times.org/simple-times/issues4-3.html> (1996).

[X.700 96] *X.700 and Other Network Management Services* [online]. Available WWW <URL: http://ganges.cs.tcd.ie/4ba2/x700/index.html> (1996).

Author Dan Plakosh, SEI
dplakosh@sei.cmu.edu

Cory Vondrak, TRW, Redondo Beach, CA

External Reviewer(s) Craig Meyers, SEI

Last Modified 10 Jan 97

Simplex Architecture

COMPLETE

Purpose and Origin

The Simplex architecture is a system integration and evolution framework that addresses the need to incorporate— incrementally and dependably— new technologies and new COTS components into long life cycle mission-critical systems, in spite of errors that could be introduced by the upgrade [Sha 96].

Technical Detail

Software is pervasive within the critical systems that form the infrastructure of modern society, both military and civilian. These systems are often large and complex and require periodic and extensive upgrading. The important technical problems include the following:

- *Black box testing problem.* Highly reliable control systems have typical Mean-Time-To-Failure (MTTF) requirements on the order of years and in some cases hundreds of years. However, the MTTF of a COTS operating system, treated as a black box with both reported and unknown bugs, is generally established via testing. Due to the lack of time for prolonged tests, the experimentally-determined MTTF is often in the range of days, or at best months. To use a COTS operating system in a system with high reliability requirements, we must find a way to bridge the reliability gap.
- *Vendor driven upgrade problem.* COTS components have a short life cycle (roughly one year.) DoD platforms change at a much slower rate and typically have longer life cycles (often 25-30 years or more). This causes the DoD platform to be susceptible to the problem that occurs when the vendor releases a new version of the COTS component. The upgrade can either be ignored or incorporated into the system. Both choices cause difficulties. Ignoring the upgrade will eventually result in a system which is burdened with unsupported and obsolete components. If, however, the upgrade is not ignored it is essentially forcing the DoD platform to change on a schedule determined by the vendor, not by the developer, maintainer, or customer. New releases usually add features and fix existing bugs. In the process they also often introduce new bugs. So upgrading is risky; a way to manage the risk is needed.
- *Upgrade paradox*: Existing fault-tolerant computing paradigms are based on replication and majority voting. The use of these techniques results in a paradox. If we upgrade only the minority of the replicated components, they will be out-voted by the majority. There will be no effect, regardless of the quality of the upgraded software. Conversely, if we upgrade the majority and make a mistake, the whole system may fail.

Collectively, these technical problems present a formidable challenge to the developers and maintainers of long life cycle systems. The Simplex

architecture is a framework for system integration and evolution. It integrates a number of technologies in its middleware service to assist users:

- technologies for integrated availability and reliability management that allow lower reliability COTS components to be used in systems demanding a high degree of reliability
- technologies for replacing software modules at runtime without having to shutdown and restart the system
- technologies that allow the system to maintain the existing level of performance in spite of potential errors in newly-replaced components
- technologies for flexible communication that allow components to dynamically publish and subscribe to needed information [Rajkumar 95]
- technologies for real-time computing (Rate Monotonic Scheduling (see pg. 313)), so that components can be replaced or modified in real time, transparently to the applications, while still meeting deadlines

Figure 26 is a highly simplified view of the data flow in a system using the Simplex architecture. Suppose *Old* is legacy software designed to control the device. *Old* has known performance characteristics and presumably, due to long use, is relatively bug free. Suppose *New* is a new version of the software with improved performance characteristics, but possibly also containing bugs since it has not been used extensively before.

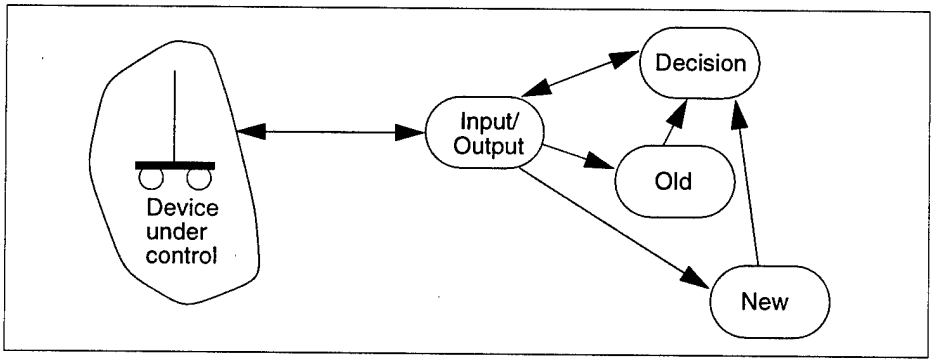


Figure 26: Simplex Architecture: Simplified Data Flow

The device under control is sampled at a regular interval by the *Input/Output* module. The data is processed by both *Old* and *New*. Instead of controlling the device directly, both modules send their results to *Decision* which, as long as *New* is behaving properly, will pass its output on to *Input/Output,* which will transmit it to the device. Should *Decision* decide that *New* is not behaving correctly, it uses the output from *Old* in-

stead. Thus the device will not perform worse than it did before the upgrade to *New* occurred. Not shown, for reasons of complexity, is the module that would actually remove a failed *New* from the system and allow it to be replaced with a corrected version for another try.

Usage Considerations

The Simplex architecture is most suitable for systems that have high availability and reliability requirements.

Since the Simplex architecture is relatively immature, pilot studies will be needed to determine its suitability for any intended application. This would involve developing a rapid prototype, using the Simplex architecture, of a simplified instance of the intended application.

Maturity

The safe, online upgrade of both software and hardware, including COTS components using the Simplex architecture has been successfully demonstrated in the laboratory. The Simplex architecture is currently being transitioned into practice via four Department of Defense (DoD) pilot studies:

1. *NSSN (new attack submarine program)* A US Navy program whose goal is the development, demonstration, and transition of a COTS-based fault-tolerant submarine control system that can be upgraded inexpensively and dependably.
2. *ISC (intelligent shipboard control program).* A US Navy program whose goal is the development of a system for fault-tolerant ship control and damage control.
3. *EDCS (evolutionary design of complex software program).* An Air Force/DARPA program whose goal is to evaluate the possible use of the Simplex architecture in the context of onboard avionics systems.
4. *JSF (joint strike force) program.* The goal is to evaluate the use of the Simplex architecture as a means for migrating legacy software to Ada 95.

Although the Simplex architecture has been designed to reduce the life-cycle cost of systems, data on its impact on system life-cycle cost is not available at this time.

Costs and Limitations

The Simplex architecture is designed to support the evolution of mission-critical systems that have a high degree of availability or reliability requirements. It will most likely not be suitable for management information systems (MIS) applications that do not have such requirements. The Simplex architecture is a COTS component-based system and the up-front investment is rather modest in the context of mission critical systems. The architecture requires modern hardware (e.g., Power PC or Pentium-based systems) and a real-time operating systems (OS) such

as POSIX.1b or an Ada runtime environment. However, legacy hardware and software can be migrated to Simplex architecture incrementally.

Complementary Technologies

Software and hardware reliability modeling and analysis allow users to estimate the impact of Simplex architecture on system reliability. System life-cycle cost estimation techniques will allow users to estimate the cost impact. Advances in fault-tolerant computing, real-time computing, OS, and network technologies will help improve the Simplex architecture since it is a framework that integrates these technologies.

Index Categories

Name of Technology	Simplex Architecture
Application category	Reapply Software Life Cycle (AP.1.9.3), Re-engineering (AP.1.9.5), Software Architecture (AP.2.1), Restart/Recovery (AP.2.10)
Quality measures category	Availability/Robustness (QM.2.1.1), Reliability (QM.2.1.2), Safety (QM.2.1.3), Real-time Responsiveness/Latency (QM.2.2.2), Maintainability (QM.3.1)
Computing reviews category	not available

References and Information Sources

[Sha 92] Sha, L.; Rajkumar, R.; & Gagliardi, M. *A Software Architecture for Dependable and Evolvable Industrial Computing Systems* (CMU/SEI-95-TR-005). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1995.

[Sha 96] Sha, L.; Rajkumar, R.; & Gagliardi, M. "Evolving Dependable Real Time Systems," 335-346. *Proceedings of the 1996 IEEE Aerospace Applications Conference.* Aspen, CO, February 3-10, 1996. New York, NY: IEEE Computer Society Press, 1996.

[Rajkumar 95] Rajkumar, R.; Gagliardi, M.; & Sha, L. "The Real-Time Publisher/Subscriber Inter-Process Communication Model for Distributed Real-Time Systems: Design and Implementation," 66-75. *The First IEEE Real-Time Technology and Applications Symposium.* Chicago, IL, May 15-17, 1995. Los Alamitos, CA: IEEE Computer Society Press, 1995.

Author

Charles B. Weinstock, SEI
weinstoc@sei.cmu.edu

Lui R. Sha, SEI
lrs@sei.cmu.edu

External Reviewer(s)	John Lehoczky, Professor, Statistics Department, CMU
Last Modified	10 Jan 97

Software Inspections

COMPLETE

Purpose and Origin

Software inspections are a disciplined engineering practice for detecting and correcting defects in software artifacts and preventing their leakage into field operations. Software inspections were introduced at IBM in the 1970s by Michael Fagan, who pioneered their early adoption and later evolution [Fagan 76, Fagan 86]. By detecting defects early and preventing their leakage into subsequent activities, the higher cost of later detection and rework is eliminated; this is essential for reduced cycle time and lower cost. Software inspections provide value in improving *reliability*, *availability*, and *maintainability*.

Technical Detail

Software inspections are strict and close examinations conducted on specifications, design, code, test, and other artifacts [Ebenau 94]. The practice of Software Inspections is composed of four elements: the structured review process, defined roles of participants, system of checklists, and forms and reports [O'Neill 88].

1. The structured review process is a systematic procedure integrated with the activities of the life cycle model. The process is composed of planning, preparation, entry criteria, conduct, exit criteria, reporting, and follow-up.
2. Software inspections are a review activity performed by peers playing the defined roles of moderator, recorder, reviewer, reader, and producer. Each role carries with it the specific behaviors, skills, and knowledge needed to achieve the expert practice of software inspections [Freedman 90].
3. A system of checklists governs each step in the structured review process and the review of the product itself, objective by objective. Process checklists are used as a guide for each activity of the structured review process. Product checklists house the strongly preferred indicators that establish the completion criteria for the organization's software products. For example, these indicators include completeness, correctness, and style:
    - Completeness is based on traceability of the requirements to the code, which is essential for maintainability.
    - Correctness is a process for reasoning about the logic and relationships of data and operations and is based on the clear specification of intended function and its faithful elaboration in code, which is essential for reliability and availability [Linger 79].
    - Style is based on consistency of recording, which is essential for maintainability.
4. Forms and reports provide uniformity in recording issues at all software inspections, reporting the results to management, and building a database useful in process management.

Savings result from early detection and correction of defects, thus avoiding the increased cost that comes with detection and correction (rework) later in the life cycle. An undetected defect that escapes detection and leaks to the next phase will likely cost at least ten times more to detect and correct than if it is found and fixed in the phase where it is originated. IBM Rochester, the 1990 winner of the Malcolm Baldrige Award, reported that defects leaking from code to test cost nine times more to detect and correct, and defects leaking from test to the field cost thirteen times more [Lindner 94].

An example may help illustrate why a leaked defect costs more. A code defect that leaks into testing may require multiple test executions to confirm the error and additional executions to obtain debug information. Once a leaked defect has been detected, the producing programmer must put aside the task at hand, refocus attention on correcting the defect and confirming the correction, and then return to the task at hand.

In addition to cost savings, the adopting organization benefits by improved predictability in cost and schedule performance, reduced defects in the field, increased customer satisfaction, and improved morale among practitioners.

The return on investment for software inspections is defined as

> Net Savings divided by Detection Cost [O'Neill 96], where:
>
> Net Savings is Cost Avoidance less Cost to Repair Now
>
> Detection Cost is the cost of preparation and the cost of conduct effort (see Costs and Limitations (pg. 354) for further elaboration).

The National Software Quality Experiment [O'Neill 95, O'Neill 96a] reveals that the return on investment (Net Savings/Detection Cost) for software inspections ranges from four to eight, independent of the context of usage.

Usage Considerations

While software inspections originated and evolved in a new development context, their usefulness in maintenance is now well established. Certain measurements obtained during software inspections reflect this context of use. For example, the lines of code inspected per conduct hour range from 250 to 500 for new development and from 1000 to 1500 for maintenance [O'Neill 95, O'Neill 96a].

The organization adopting software inspections practice seeks to prevent defect leakage. Following training, the organization can expect to detect 50% of the defects present. It may take twelve to eighteen months to

achieve expert practice, where defect detection is expected to range from 60% to 90%. IBM reported 83% and ATT reported 92% defect detection resulting from software inspections practice [O'Neill 89].

The adoption of software inspections practice is competency-enhancing and typically meets little resistance among practitioners.

Maturity

The maturity of a technology can be reasoned about in terms of its long-term, widespread use in a variety of usage domains and its transition from early adopters through late adopters. Software inspections have twenty-five years of application and evolution. They are known to deliver added economic value.

Software inspections are a rigorous form of peer reviews, a key process area of the SEI Capability Maturity Model[1] [Paulk 95, Humphrey 89]. Even though peer reviews are assigned to level 3 in the software process maturity framework and many organizations limit their software process improvement agenda to the key process areas for the maturity level they are seeking to achieve, the population of successful software inspections adopters ranges from level 1 to 5.

While a level 3 organization is expected to have a more mature software engineering capability with defined life cycle activities that use software inspections to verify exit criteria, the early adoption of software inspections practice stimulates the improvements in software engineering practice necessary to achieve level 3. In addition, the early adoption of software inspections brings with it some beneficial side effects, such as cross pollination of ideas, ability to work together, and team building.

The widespread use and variety of usage domains is best illustrated by the National Software Quality Experiment [O'Neill 95,96a], which has been gathering data about software defects and inspections practice. In this study, thousands of participants from dozens of organizations are populating the experiment database with thousands of defects of all types, along with pertinent information needed to pinpoint their root causes. The range of analysis bins identified in the experiment includes software process maturity level (1,2), organization type (government, Department of Defense industry, commercial), product type (embedded, organic), programming language (old style, modern), and global region (North America, Pacific Rim).

1. Capability Maturity Model and CMM are service marks of Carnegie Mellon University.

Costs and Limitations

The rollout and operating costs associated with software inspections include

- the initial training of practitioners and managers
- the ongoing preparation and conduct of inspection sessions
- the ongoing management and use of measurement data for defect prevention and return on investment computations

Initial training. To adopt software inspections practice properly, each participant is trained in the structured review process, roles of participants, system of process and product checklists, and forms and reports. The cost to acquire the knowledge, skills, and behaviors is twelve hours per practitioner [O'Neill 89]. In addition, each manager is trained in the responsibilities for rolling out the technology and in the interpretation and use of measurements taken. The management training takes four hours.

Preparation and conduct of inspection sessions. The cost of performing software inspections includes the preparation effort of each participant before the session and the conduct effort of participants in the inspections session. Typically five people participate and each expends one to two hours of preparation and one to two hours of conduct. This cost of 10 to 20 hours of total effort per session typically results in the early detection of five to ten defects in 250-500 lines of new development code or 1000-1500 lines of legacy code [O'Neill 95, O'Neill 96a].

Management and use of measurement data. Three steps are involved in the management and use of measurement data:

1. The software inspections database structure for the organization is established. The cost to establish a database structure and produce the basic user macros to operate on the data is two person-months.
2. Measurement results are entered, thus populating the database structure. The cost to populate the database with measured results is included in the cost of performing software inspections above, where the recorder for each inspection session is responsible for entering the session data into the software inspections database.
3. Operations on the measurement database generate derived metrics in the form of reports and graphs. The cost to generate reports and graphs on the derived metrics is one person-day per month.

Dependencies

For an organization to obtain the full benefits of software inspections, a defined process for software product engineering must be in place. This will permit software inspections to be used in the practice of statistical process control. In this context, software inspections provide the exit criteria for each life-cycle activity. Furthermore, the completion criteria for each type of artifact is specified and used in practice.

Alternatives

Alternatives include software walkthroughs, a less rigorous form of peer reviews. Walkthroughs may cost as much as inspections, but they deliver less. Walkthroughs are producer-led reviews whose results are not recorded, thus precluding the application of the statistical process control practice needed to advance software process maturity.

Complementary Technologies

Cyclomatic complexity (see pg. 145) can be used to optimize the practice of software inspections on legacy code during maintenance operations. In this approach, candidates for inspection are selected (after modules are rank ordered) from those with the highest complexity ratings, where the defect density is known to be high.

This legacy code maintenance strategy can be extended by rank ordering all modules based upon defects encountered in the past year and by rank ordering the modules expected to be adapted and perfected in the coming year. Modules for inspection are then selected based on their rank ordering in cyclomatic complexity, defect history, and expected rework.

Index Categories

Name of technology	Software Inspections
Application category	Detailed Design (AP.1.3.5), Code (AP.1.4.2), Unit Testing (AP.1.4.3.4), Component Testing (AP.1.4.3.5)
Quality measures category	Correctness (QM.1.3), Reliability (QM.2.1.2), Availability (QM.2.1.1), Maintainability (QM.3.1)
Computing reviews category	Program Verification (D.2.4), Testing and Debugging (D.2.5)

References and Information Sources

[Ebenau 94] Ebenau, Robert G. & Strauss, Susan H. *Software Inspection Process.* New York, NY: McGraw-Hill, 1994.

[Fagan 76] Fagan, M. "Design and Code Inspections to Reduce Errors in Program Development." *IBM Systems Journal 15*, 3 (1976): 182-211.

[Fagan 86] Fagan, M. "Advances in Software Inspections." *IEEE Transactions on Software Engineering 12*, 7 (July 1986): 744-751.

[Freedman 90] Freedman, D.P. & Weinberg, G.M. *Handbook of Walkthroughs, Inspections, and Technical Reviews.* New York, NY: Dorset House, 1990.

[Humphrey 89] Humphrey, Watts S. *Managing the Software Process.* Reading, MA: Addison-Wesley, 1989.

[Lindner 94] Lindner, Richard J. & Tudahl, D. "Software Development at a Baldrige Winner," 167-180. *Proceedings of ELECTRO '94.* Boston, MA, May 12, 1994. New York, NY: IEEE, 1994.

[Linger 79] Linger, R.C.; Mills, H.D.; & Witt, B.I. *Structured Programming: Theory and Practice.* Reading, MA: Addison-Wesley, 1979.

[O'Neill 88] O'Neill, Don & Ingram, Albert L. "Software Inspections Tutorial," 92-120. *Software Engineering Institute Technical Review 1988.* Pittsburgh, PA: Carnegie Mellon University, Software Engineering Institute, 1988.

[O'Neill 89] O'Neill, Don. *Software Inspections Course and Lab.* Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1989.

[O'Neill 92] O'Neill, Don. "Software Inspections: More Than a Hunt for Errors." *Crosstalk, Journal Of Defense Software Engineering* 30 (January 1992): 8-10.

[O'Neill 95] O'Neill, Don. "National Software Quality Experiment: Results 1992-1995." *Proceedings of the Seventh Annual Software Technology Conference.* Salt Lake City, UT, April 9-14, 1995. Hill Air Force Base, UT: Software Technology Support Center, 1995.

[O'Neill 96a] O'Neill, Don. "National Software Quality Experiment: Results 1992-1996." *Proceedings of the Eighth Annual Software Technology Conference.* Salt Lake City, UT, April 21-26, 1996. Hill Air Force Base, UT: Software Technology Support Center, 1996.

[O'Neill 96b] O'Neill, Don. *Peer Reviews Key Process Area Handbook.* Gaithersburg, MD: Don O'Neill Consulting, 1996.

[Paulk 95] Paulk, Mark C. *The Capability Maturity Model: Guidelines for Improving the Software Process.* Reading, MA: Addison-Wesley Publishing Company, 1995.

Author Don O'Neill, Don O' Neill Consulting
ONeillDon@aol.com

External Reviewer(s) Alex Elentukh, Fidelity Investments
Rick Linger, SEI
Watts Humphrey, SEI
Joan Weszka, Lockheed Martin

Last Modified 10 Jan 97

Statistical-Based Intrusion Detection — ADVANCED

Note

We recommend Intrusion Detection, pg. 217, as prerequisite reading for this technology description.

Purpose and Origin

Intrusion detection systems (IDS) automate the detection of security violations through computer processing of system audit information. One IDS approach, rule-based intrusion detection (RBID) (see pg. 331), seeks to identify intrusion attempts by matching audit data with known patterns of intrusive behavior. RBID systems rely on codified rules of known intrusions to detect intrusive behavior. Intrusion attempts not represented in an RBID rule base will go undetected by these systems. To help overcome this limitation, statistical methods have been employed to identify audit data that may *potentially* indicate intrusive or abusive behavior. Known as statistical-based intrusion detection (SBID) systems, these systems analyze audit trail data by comparing them to typical or predicted profiles in an effort to find pending or completed computer security violations. This emerging technology seeks to increase the *availability* of computer systems by automating the detection and elimination of intrusions.

Technical Detail

SBID systems seek to identify abusive behavior by noting and analyzing audit data that deviates from a predicted norm. SBID is based on the premise that intrusions can be detected by inspecting a system's audit trail data for unusual activity, and that an intruder's behavior will be noticeably different than that of a legitimate user. Before unusual activity can be detected, SBID systems require a characterization of user or system activity that is considered "normal." These characterizations, called *profiles*, are typically represented by sequences of events that may be found in the system's audit data. Any sequence of system events deviating from the expected profile by a statistically significant amount is flagged as an intrusion attempt [Sundaram 96]. The main advantage of SBID systems is that intrusions can be detected without *a priori* information about the security flaws of a system [Kremmerer 94].

SBID systems typically employ statistical anomaly and rule-based misuse models [Mukherjee 94]. System profiles, user profiles, or both may be used to define expected behavior. User profiles, if used, are specific to each user and are dynamically maintained. As a user's behavior changes over time, so too will his user profile. No such profiles are used in RBID systems. As is the case with RBID systems, known intrusion scenarios can be codified into the rule base of SBID systems.

Interesting variations on this theme include the following:

- Predictive pattern generation, which uses a rule base of user profiles defined as statistically-weighted event sequences [Teng 90]. This method of intrusion detection attempts to predict future events based on events that have already occurred. Advantages of this approach include its ability to detect misuse as well as intrusions and its ability to detect and respond quickly to anomalous behavior.
- Connectionist approaches in which neural networks are used to create and maintain behavior profiles [Lunt 93]. Advantages of neural approaches include their ability to cope with noisy data and their ability to adapt to new user communities. Unfortunately, trial and error is required to train the net, and it is possible for an intruder to train the net during its learning phase to ignore intrusion attempts [Sundaram 96].

Usage Considerations

An advantage of SBID systems is that they are able to adaptively learn the behavior of the users they monitor and are thus potentially more sensitive to intrusion attempts than are humans [Sundaram 96, Lunt 93]. However, SBID systems require the creation and maintenance of user/system profiles. These profiles are sensitive to hardware and software modifications, and will need to be updated whenever the system or network they used to protect is modified. Additional work is required to determine how statistical user/system profiles should be created and maintained [Lunt 93].

Maturity

Statistical intrusion detection algorithms have been in existence since at least 1988. Several prototype systems have been developed, including Haystack [Smaha 88], IDES [Lunt 93], and MIDAS [Mukherjee 94]. MIDAS is a deployed real-time SBID that provides security protection for the National Computer Center's networked mainframe computer. IDES, which is deployed at both SRI and FBI locations, is an IDS that combines SBID with RBID to detect a wider range of intrusion attempts. Another deployed security system containing aspects of SBID technology is AT&T Bell Lab's Dragons system which protects their Internet gateway;[1] the Dragons system has succeeded in detecting intrusion attempts ranging from attempted "guest" logins to forged NFS packets [Mukherjee 94].

Costs and Limitations

In addition to the costs associated with creating audit trails and maintaining user profiles, there are several risks and limitations associated with SBID technology:

- Because user profiles are updated periodically, it is possible for an insider to slowly modify his behavior over time until a new behavior

1. See http://www.research.att.com for more details.

pattern has been established within which an attack can be safely mounted [Lunt 93].

- Determining an appropriate threshold for "statistically significant deviations" can be difficult. If the threshold is set too low, anomalous activities that are not intrusive are flagged as intrusive (false positive). If the threshold is set too high, anomalous activities that are intrusive are not flagged as intrusive (false negative).
- Defining user profiles may be difficult, especially for those users with erratic work schedules/habits.

Like RBID systems, SBID systems will negatively affect throughput because of to the need to collect and analyze audit data. However, in contrast with RBID systems, SBID systems do not always lag behind the intruders. Detection of anomalous behavior, whether or not it is codified as a known intrusion attempt, may be sufficient grounds for an SBID system to detect an intruder.

Use of this technology requires personnel who are experienced in statistics and intrusion detection techniques and who have in-depth knowledge of audit collection mechanisms.

Dependencies

Expert systems are an enabler for this technology.

Alternatives

Other approaches to intrusion detection include model-based or rule-based approaches (see pg. 331), and approaches based on genetic algorithms. Manual examination of recorded audit data and online monitoring of access activity by knowledgeable personnel are the only other known alternatives.

Complementary Technologies

Rule-based intrusion detection systems (see pg. 331) can be used in conjunction with statistical-based intrusion detection systems to catch a wider variety of intrusion attempts, and user authentication systems can be used to help verify user identify.

Index Categories

Name of technology	Statistical-Based Intrusion Detection
Application category	System Security (AP.2.4.3)
Quality measures category	Security (QM.2.1.5)
Computing reviews category	Operating Systems Security and Protection (D.4.6), Computer-Communication Networks Security and Protection (C.2.0), Security and Protection (K.6.5)

References and Information Sources

[Bell 76] Bell, D. E. & LaPadula, L. J. *Secure Computer System: Unified Exposition and Multics Interpretation* Rev. 1 (MTR-2997). Bedford, MA: MITRE Corp., 1976.

[Kemmerer 94] Kemmerer, Richard A. "Computer Security," 1153-1164. *Encyclopedia of Software Engineering.* New York, NY: John Wiley and Sons, 1994.

[Lunt 93] Lunt, Teresa F. "A Survey of Intrusion Detection Techniques." *Computers and Security 12*, 4 (June 1993): 405-418.

[Mukherjee 94] Mukherjee, Biswanath, L.; Heberlein, Todd; & Levitt, Karl N. "Network Intrusion Detection." *IEEE Network 8,* 3 (May-June 1994): 26-41.

[Smaha 88] Smaha, Stephen E. "Haystack: An Intrusion Detection System," 37-44. *Proceedings of the Fourth Aerospace Computer Security Applications Conference.* Orlando, Florida, December 12-16, 1988. Washington, DC: IEEE Computer Society Press, 1989.

[Spafford 88] Spafford, Eugene H. *The Internet Worm Program: An Analysis* (CSD-TR-823). West Lafayette, IN: Purdue University, 1988.

[Sundaram 96] Sundaram, Aurobindo. *An Introduction to Intrusion Detection* [online]. Available WWW <URL:http://www.acm.org/crossroads/xrds2-4/xrds2-4.html>

[Teng 90] Teng, Henry S.; Chen, Kaihu; & Lu, Stephen C. "Security Audit Trail Analysis Using Inductively Generated Predictive Rules," 24-29. *Sixth Conference on Artificial Intelligence Applications.* Santa Barbara, CA, May 5-9, 1990. Los Alamitos, CA: IEEE Computer Society Press, 1990.

Author Mark Gerken, Rome Laboratory
gerken@ai.rl.af.mil

Last Modified 10 Jan 97

TAFIM Reference Model

ADVANCED

Note

We recommend Reference Models, Architectures, Implementations—An Overview, pg. 319, as prerequisite reading for this technology.

Purpose and Origin

The Technical Architectural Framework for Information Management (TAFIM) reference model was developed by the Defense Information Systems Agency (DISA) to guide the evolution of Department of Defense (DoD) systems, including sustaining base, strategic, and tactical systems, as well as interfaces to weapon systems. Application of the TAFIM reference model is required on most DoD systems [Paige 93]. TAFIM is a set of services, standards, design components, and configurations that are used in design, implementation, and enhancement of information management system architectures. The intent is that the DoD infrastructure will have a common architecture that will, over time, be a fully *flexible* and *interoperable* enterprise. Details on the TAFIM model are available in a seven volume TAFIM document, but are primarily in Volume 3 [TAFIM 94].

Technical Detail

The TAFIM reference model (Figure 27) describes services (functionality) needed within each of the model's components. It contains a set of general principles on how components and component services relate to each other. This model is designed to enhance transition from legacy applications to a distributed environment. TAFIM addresses the following six software components:

1. *Application software.* Application software consists of mission area applications and support applications. Mission area applications may be custom-developed software, commercial-off-the-shelf (COTS) products, or Non-developmental items (NDI). Support applications are building blocks for mission area applications. They manage processing for the communication environment and can be shared by multiple mission and support applications. Common COTS support applications include multimedia, communications, business processing, environment management, database utilities, and engineering support (analysis, design, modeling, development, and simulation) capabilities.
2. *Application platform.* Application platform consists of hardware services and software services, including operating system, real-time monitoring program, and peripheral drivers. Application software must access platform resources by a request across application program interfaces (APIs) (see pg. 79) to ensure integrity and consistency. A platform service may be realized by a single process shared by a group of applications, or by a distributed system with portions of an

application operating on separate processors. Application platform services include software engineering, user interface, data management, data interchange, graphic, network, and operating system capabilities.

3. *Application platform cross-area services.* Application platform cross-area services are services that have a direct effect on the operation of one or more of the functional areas. Application platform cross-area services include culturally-related application environments, security, system administration and distributed computing capabilities.

4. *External environment.* The external environment supports system and application interoperability and user and data portability. The external environment interface specifies a complete interface between the application platform and underlying external environment. The external environment includes human-computer interaction, information services, and communication capabilities.

5. *TAFIM application program interface (API).*The API is the interface between an application and a service that resides on a platform. The API specifies how a service is invoked— without specifying its implementation— so that the implementation may be changed without causing a change in the applications that use that API. The API makes the platform transparent to the application. A platform may be a single computer or a network of hosts, clients, and servers where distributed applications are implemented. A service invoked through an API can reside on the same platform as the requesting application, on a different platform, or on a remote platform. APIs are defined for mission and support applications and platform services. APIs are generally required for platform services such as compilers, window management, data dictionaries, database management systems, communication protocols, and system management utilities.

6. *TAFIM external environment interface.* The TAFIM external environment interface (which could be considered and API) is between the application platform and the external environment. This interface allows the exchange of information. It supports system and application software interoperability. User and data portability are directly provided by the external environment interface.

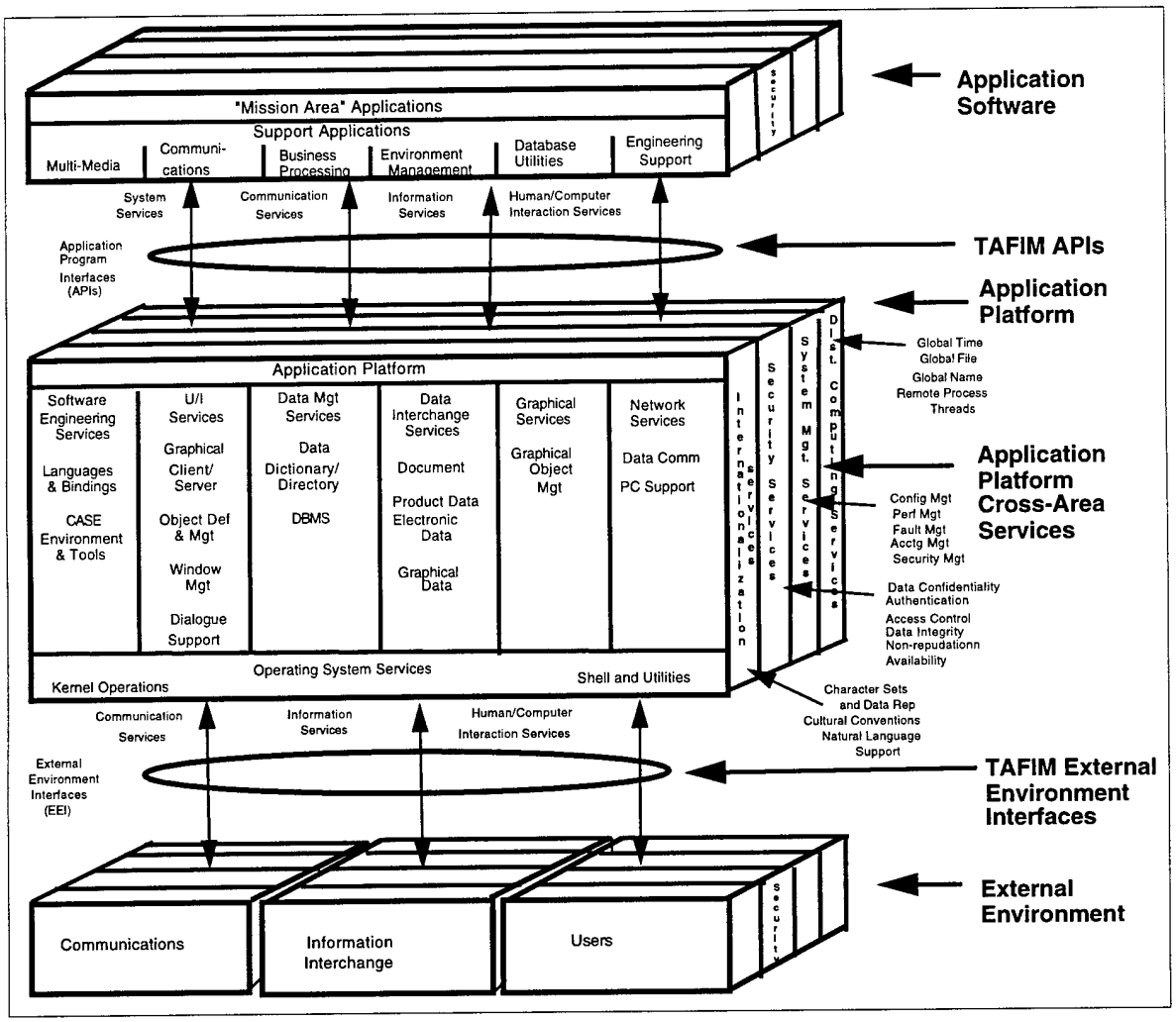


Figure 27: DoD TAFIM Technical Reference Model

Usage Considerations

The TAFIM reference model is applicable to most information systems, including sustaining base, strategic, and tactical systems, as well as interfaces to weapon systems [TAFIM 94]. It is mandatory for use on most DoD programs [Paige 93]. However, systems built using the reference model have been criticized by Rear Adm. John Gauss, the Interoperability Chief at DISA, when speaking on systems in the field in Bosnia: "We have built a bunch of state-of-the-art, open-systems, TAFIM-compliant stove-pipes" [Temin 96]. TAFIM-compliant means that the applicable standards and guidelines are met for the implemented component services. This suggests that even when complying with the TAFIM reference model, problems of interoperability are not necessarily resolved. The Joint Technical Architecture (JTA) provides a set of standards and guidelines for C4I systems, specifically in the area of interoperability, that supersedes TAFIM Volume 7 [JTA 96].

There are TAFIM-compliant software products available for use when implementing a TAFIM-based architecture in areas such as support applications, communication services, business process services, environment management, and engineering services. Additional products exist or are being developed in areas such as user interface, data management, data interchange, graphics, operating systems, internationalization, security system management, and distributed computing.

Maturity

The latest version of TAFIM, Version 2.0, was published in 1994. DoD organizations and contractors have been applying this set of guidelines to current and future information systems. The Defense Information Infrastructure (DII) Common Operating Environment (COE) (see pg. 155) is an implementation of TAFIM. This COE is currently being used by the Global Command and Control System (GCCS) and the Global Combat Support System (GCSS). The Air Force Theater Battle Management Core System (TBMCS) is also required to comply with the TAFIM and use the COE. It may take several years, after multiple new TAFIM-compliant systems are in the field, to determine the effectiveness of the reference model with respect to achieving a common, flexible, and interoperable DoD infrastructure.

Costs and Limitations

The TAFIM reference model does not fully specify components and component connections [Clements 96]. It does not dictate the specific components for implementation. (No reference model prescribes implementation solutions.) TAFIM does provide the guidance necessary to improve commonality among DoD information technical architectures.

One contractor has found that there is no cost difference in using the TAFIM reference model (as compared to any other reference model) when designing and implementing a software architecture. This is based on the fact that application of a reference model is part of the standard design and implementation practice.

Dependencies

The TAFIM reference model is dependent on the evolution of component and service standards that apply specifically to software; it may be affected by computer platforms and network hardware as well.

Alternatives

Under conditions where the TAFIM reference model is not required, an alternative model would be the Reference Model for Frameworks of Software Engineering Environments (known as the ECMA model [ECMA 93]) that is promoted in Europe and used commercially and worldwide. Commercially-available Hewlett-Packard products use this model [HP 96]. Another alternative would be the Common Object Request Broker Architecture (CORBA) if the design called for object-oriented infrastructure (see pg. 107).

Complementary Technologies

Open systems (see pg. 135) would be a complementary technology to TAFIM because work done in open system supports the TAFIM goals of achieving interoperable systems.

Index Categories

Name of technology	TAFIM Reference Model
Application category	Software Architecture Models (AP.2.1.1), Distributed Computing (AP.2.1.2)
Quality measures category	Maintainability (QM.3.1), Interoperability (QM.4.1)
Computing reviews category	Distributed Systems (C.2.4), Software Engineering Design (D.2.10)

References and Information Sources

[Clements 96] Clements, P. & Northrop, L. *Software Architecture: An Executive Overview* (CMU/SEI-96-TR-003). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1996.

[ECMA 93] *Reference Model for Frameworks of Software Engineering Environments* 3rd Edition (NIST Special Publication 500-211/ECMA TR/55). Prepared jointly by NIST and ECMA. Washington, DC: U.S. Government Printing Office, 1993.

[HP 96] *Integrated Solutions Catalog for the SoftBench Product Family*. Palo Alto, CA: Hewlett-Packard, 1993.

[JTA 96] *Joint Technical Architecture* [online]. Available WWW <URL: http://www.itsi.disa.miil.jta/html> (1996).

[Paige 93] Paige, Emmett. *Selection of Migration Systems* ASD (C3I) Memorandum. Washington, DC: Department of Defense, November 12, 1993.

[TAFIM 94] U.S. Department Of Defense. *Technical Architecture Framework For Information Management (TAFIM)* Volumes 1-8, Version 2.0. Reston, VA: DISA Center for Architecture, 1994. Also available [online] WWW <URL: http://www.jcdbs.itsi.disa.mil:8000/ces/tafim> (1996).

[Temin 96] Temin, Thomas, ed. "Mishmash at Work (DoD Systems in Bosnia are not Interoperable)." *Government Computer News 15, 7* (April 1996): 28.

Author

Darleen Sadoski, GTE
sadoski.darleen@mail.ndhm.gtegsc.com

External Reviewer(s)

Peter Garrabrant, GTE
Tricia Oberndorf, SEI

Last Modified 10 Jan 97

Three Tier Software Architectures

COMPLETE

Note

We recommend Client/Server Software Architectures, pg. 101, as prerequisite reading for this technology description.

Purpose and Origin

The three tier software architecture emerged in the 1990s to overcome the limitations of the two tier architecture (see pg. 381). The third tier (middle tier server) is between the user interface (client) and the data management (server) components. This middle tier provides process management where business logic and rules are executed and can accommodate hundreds of users (as compared to only 100 users with the two tier architecture) by providing functions such as queuing, application execution, and database staging. The three tier architecture is used when an effective distributed client/server design is needed that provides (when compared to the two tier) increased *performance*, *flexibility*, *maintainability*, *reusability*, and *scalability*, while hiding the complexity of distributed processing from the user. For detailed information on three tier architectures see Schussel and Eckerson. Schussel provides a graphical history of the evolution of client/server architectures [Schussel 96, Eckerson 95].

Technical Detail

A three tier distributed client/server architecture (as shown in Figure 28) includes a user system interface top tier where user services (such as session, text input, dialog, and display management) reside.

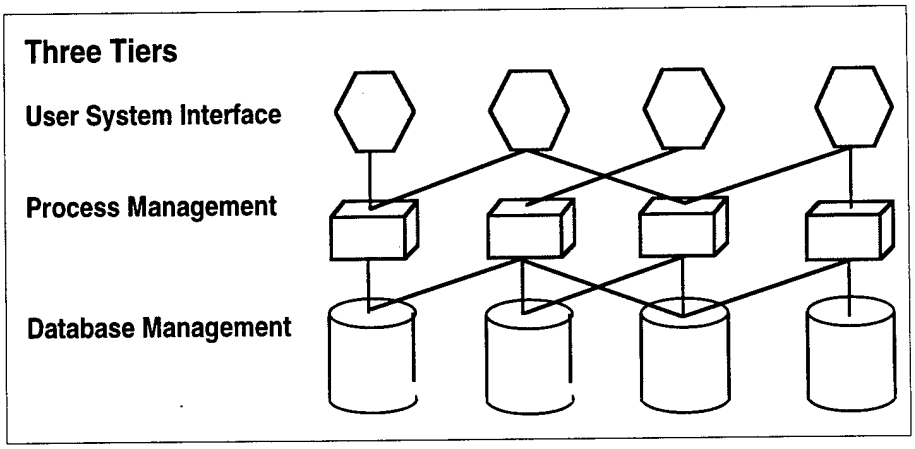


Figure 28: Three tier distributed client/server architecture depiction [Louis 95]

The middle tier provides process management services (such as process development, process enactment, process monitoring, and process resourcing) that are shared by multiple applications. The third tier provides database management functionality and is dedicated to data and file ser-

vices that can be optimized without using any proprietary database management system languages. The data management component ensures that the data is consistent throughout the distributed environment through the use of features such as data locking, consistency, and replication. It should be noted that connectivity between tiers can be dynamically changed depending upon the user's request for data and services.

The middle tier server (also referred to as the application server) improves performance, flexibility, maintainability, reusability, and scalability by centralizing process logic. Centralized process logic makes administration and change management easier by localizing system functionality so that changes must only be written once and placed on the middle tier server to be available throughout the systems. With other architectural designs, a change to a function (service) would need to be written into every application [Eckerson 95].

In addition, the middle process management tier controls transactions and asynchronous queuing to ensure reliable completion of transactions [Schussel 96]. The middle tier manages distributed database integrity by the two phase commit process (see pg. 151). It provides access to resources based on names instead of locations, and thereby improves scalability and flexibility as system components are added or moved [Edelstein 95].

It should be noted that recently, mainframes have been combined as servers in distributed architectures to provide massive storage and improve security (see pg. 227).

Usage Considerations

Three tier architectures are used in commercial and military distributed client/server environments in which shared resources, such as heterogeneous databases and processing rules, are required [Edelstein 95]. The three tier architecture will support hundreds of users, making it more scalable than the two tier architecture (see pg. 381) [Schussel 96].

Three tier architectures facilitate software development because each tier can be built and executed on a separate platform, thus making it easier to organize the implementation. Also, three tier architectures readily allow different tiers to be developed in different languages, such as a graphical user interface language for the top tier; C, C++, SmallTalk, Basic, Ada 83 (see pg. 61), or Ada 95 (see pg. 67) for the middle tier; and SQL for much of the database tier [Edelstein 95].

Migrating a legacy system to a three tier architecture can be done in a manner that is low-risk and cost-effective. This is done by maintaining the old database and process management rules so that the old and new

systems will run side by side until each application and data element or object is moved to the new design. This migration might require rebuilding legacy applications with new sets of tools and purchasing additional server platforms and service tools, such as transaction monitors (see pg. 373) and message-oriented middleware (see pg. 247). The benefit is that three tier architectures hide the complexity of deploying and supporting underlying services and network communications.

Maturity

Three tier architectures have been used successfully since the early 1990s on thousands of systems of various types throughout the Department of Defense (DoD) and in commercial industry, where distributed information computing in a heterogeneous environment is required. An Air Force system that is evolving from a legacy architecture to a three tier architecture is Theater Battle Management Core System (TBMCS).

Costs and Limitations

Building three tier architectures is complex work. Programming tools that support the design and deployment of three tier architectures do not yet provide all of the desired services needed to support a distributed computing environment.

A potential problem in designing three tier architectures is that separation of user interface logic, process management logic, and data logic is not always obvious. Some process management logic may appear on all three tiers. The placement of a particular function on a tier should be based on criteria such as the following [Edelstein 95]:

- ease of development and testing
- ease of administration
- scalability of servers
- performance (including both processing and network load)

Dependencies

Database management systems must conform to X/Open systems standards and XA Transaction protocols to ensure distributed database integrity when implementing a heterogeneous database two phase commit.

Alternatives

Two tier client server architectures (see pg. 381) are appropriate alternatives to the three tier architectures under the following circumstances:

- when the number of users is expect to be less than 100
- for non-real-time information processing in non-complex systems that requires minimal operator intervention

Distributed/collaborative enterprise computing (see pg. 163) is seen as a viable alternative, particularly if object-oriented technology on an enter-

prise-wide scale is desired. An enterprise-wide design is comprised of numerous smaller systems or subsystems.

Complementary Technologies

Complementary technologies to three tier architectures are object-oriented design (to implement decomposable applications) (see pg. 283), three tier client/server architecture tools, and database two phase commit processing (see pg. 151).

Index Categories

Name of technology	Three Tier Software Architectures
Application category	Client/Server (AP.2.1.2.1)
Quality measures category	Maintainability (QM.3.1), Scalability (QM.4.3), Reusability (QM.4.4), Reliability (QM.2.1.2)
Computing reviews category	Distributed Systems (C.2.4), Software Engineering Design (D.2.10)

References and Information Sources

[Dickman 95] Dickman, A. "Two-Tier Versus Three-Tier Apps." *Informationweek* 553 (November 13, 1995): 74-80.

[Eckerson 95] Eckerson, Wayne W. "Three Tier Client/Server Architecture: Achieving Scalability, Performance, and Efficiency in Client Server Applications." *Open Information Systems 10,* 1 (January 1995): 3(20).

[Edelstein 95] Edelstein, Herb. "Unraveling Client Server Architectures." *DBMS 7,* 5 (May 1994): 34(7).

[Gallaugher 96] Gallaugher, J. & Ramanathan, S. "Choosing a Client/Server Architecture. A Comparison of Two-Tier and Three-Tier Systems." *Information Systems Management Magazine 13,* 2 (Spring 1996): 7-13.

[Louis 95] *Louis* [online]. Available WWW <URL: http://www.softis.is> (1995).

[Newell 95] Newell, D.; Jones, O.; & Machura, M. "Interoperable Object Models for Large Scale Distributed Systems," 152+32. *Proceedings. International Seminar on Client/Server Computing.* La Hulpe, Belgium, October 30-31, 1995. London, UK: IEE, 1995.

[Schussel 96] Schussel, George. *Client/Server Past, Present, and Future* [online]. Available WWW <URL: http://www.dciexpo.com/geos/> (1995).

Author

Darleen Sadoski, GTE
sadoski.darleen@mail.ndhm.gtegsc.com

External Reviewer(s)	Paul Clements, SEI Frank Rogers, GTE
Last Modified	10 Jan 97

Transaction Processing Monitor Technology ADVANCED

Note

We recommend Client/Server Software Architectures, pg. 101, as prerequisite reading for this technology description.

Purpose and Origin

Transaction processing (TP) monitor technology provides the distributed client/server environment the capacity to *efficiently* and *reliably* develop, run, and manage transaction applications.

TP monitor technology controls transaction applications and performs business logic/rules computations and database updates. TP monitor technology emerged 25 years ago when Atlantic Power and Light created an online support environment to share concurrently applications services and information resources with the batch and time sharing operating systems environment. TP monitor technology is used in data management, network access, security systems, delivery order processing, airline reservations, and customer service. Use of TP monitor technology is a cost-effective alternative to upgrading database management systems or platform resources to provide this same functionality. Dickman and Hudson provide more details on TP monitor technology [Dickman 95, Hudson 94].

Technical Detail

TP monitor technology is software that is also referred to as middleware (see pg. 251). It can provide application services to thousands of clients in a distributed client/server environment. TP monitor technology does this by multiplexing client transaction requests (by type) onto a controlled number of processing routines that support particular services. These events are depicted in Figure 29.

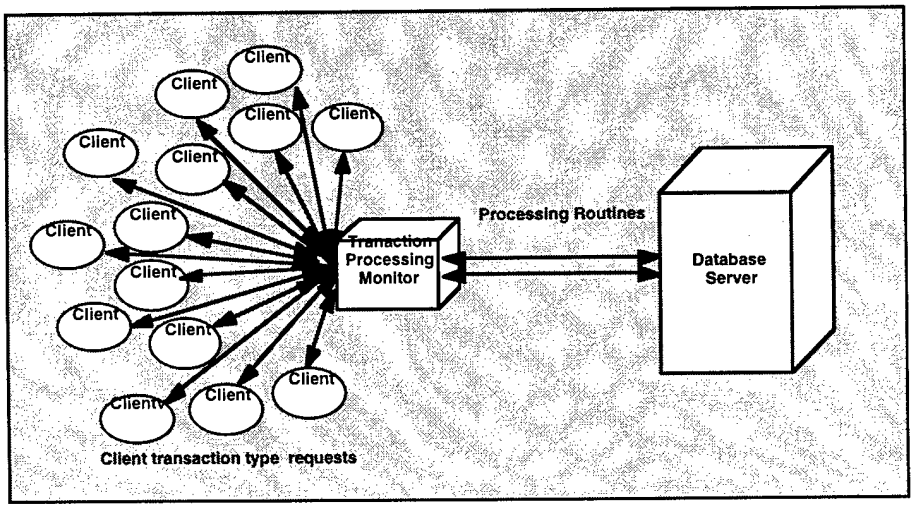


Figure 29: Transaction Processing Monitor Technology

Clients are bound, serviced, and released using stateless servers that minimize overhead. The database sees only the controlled set of processing routines as clients [Dickman 95, Hudson 94].

TP monitor technology maps numerous client requests through application services routines to improve system performance. The TP monitor technology (located as a server) can also take the application transitions logic from the client. This reduces the number of upgrades required by these client platforms. In addition, TP monitor technology includes numerous management features, such as restarting failed processes, dynamic load balancing, and enforcing consistency of distributed data. TP monitor technology is easily scalable by adding more servers to meet growing numbers of users [Dickman 95, Hudson 94].

TP monitor technology is independent of the database architecture. It supports flexible and robust business modeling and encourages modular, *reusable* procedures. TP monitor designs allow application program interfaces (APIs) (see pg. 79) to support components such as heterogeneous client libraries, databases and resource managers, and peer-level application systems. TP monitor technology supports architecture *flexibility* because each component in a distributed system is comprised of products that are designed to meet specific functionality, such as graphical user interface builders and database engines [Dickman 95, Hudson 94].

Usage Considerations

Within distributed client/server systems, each client that is supported adds overhead to system resources (such as memory). Responsiveness is improved and system resource overhead is reduced by using TP monitor technology to multiplex many clients onto a much smaller set of application service routines. TP monitor technology provides a highly active system that includes services for delivery order processing, terminal and forms management, data management, network access, authorization, and security.

TP monitor technology supports a number of program-to-program communication models, such as store-and-forward, asynchronous, remote procedure call (RPC) (see pg. 323), and conversational. This improves interactions among application components. TP monitor technology provides the ability to construct complex business applications from modular, well-defined functional components. Because this technology is well-known and well-defined it should reduce program risk and associated costs [Dickman 95, Hudson 94].

Maturity

TP monitor technology has been used successfully in the field for 25 years. TP monitor technology is used for delivery order processing, hotel

and airline reservations, electronic fund transfers, security trading, and manufacturing resource planning and control. It improves batch and time-sharing application effectiveness by creating online support to share application services and information resources [Dickman 95, Hudson 94].

Costs and Limitations

TP monitor technology makes database processing cost-effective for on-line applications. Spending relatively little money on TP monitor technology can result in significant savings compared to the resources required to improve database or platform resources to provide the same functionality [Dickman 95].

A limitation to TP technology is that the implementation code is usually written in a lower-level language (such as COBOL), and is not yet widely available in the popular visual toolsets [Shussel 96].

Alternatives

A variation of TP monitor technology is session based technology. In the TP monitor technology, transactions from the client are treated as messages. In the session based technology, a single server provides both database and transaction services. In session based technology, the server must be aware of clients in advance to maintain each client's processing thread. The session server must constantly send messages to the client (even when work is not being done in the client) to ensure that the client is still alive. Session based architectures are not as scalable because of the adverse effect on network performance as the number of clients grow.

Another alternative to TP monitor technology is remote data access (RDA). The RDA centers the application in a client computer, communicating with back-end database servers. Clients can be network-intensive, but scalability is limited.

A third alternative to TP monitor technology is the database server approach, which provides functions (usually specific to the database) and is architecturally locked to the specific database system [Dickman 95, Hudson 94].

Complementary Technologies

Complementary technologies include mainframe client/server software architectures (see pg. 227) and three tier software architectures (see pg. 367); in both cases the TP monitor technology could server as the middle tier.

Index Categories

Name of technology	Transaction Processing Monitor Technology
Application category	Client/Server (AP.2.1.2.1), Client/Server Communication (AP.2.2.1)
Quality measures category	Efficiency/ Resource Utilization (QM.2.2), Reusability (QM.4.4), Maintainability (QM.3.1)
Computing reviews category	Distributed Systems (C.2.4)

References and Information Sources

[Dickman 95] Dickman, A. "Two-Tier Versus Three-Tier Apps." *Informationweek* 553 (November 1995): 74-80.

[Framework 96] *Framework for Complex Client/Server Computing* [online]. Available WWW <URL: http://www.mch.sni.dc/public/mr/01+p2/tpmon.htm> (1996).

[Hudson 94] Hudson, D. & Johnson, J. *Client-Server Goes Business Critical.* Dennis, MA: The Standish Group International, 1994.

[Schussel 96] Schussel, George. *Client/Server Past, Present, and Future* [online]. Available WWW <URL: http://www.dciexpo.com/geos/> (1995).

[TP 96] *TP Lite vs. TP Heavy* [online]. Available WWW <URL: http://www.byte.com/art/9504/sec11/art4.htm> (1996).

Author Darleen Sadoski, GTE
sadoski.darleen@mail.ndhm.gtegsc.com

External Reviewer(s) David Altieri, GTE

Last Modified 10 Jan 97

Trusted Operating Systems — ADVANCED

Note

We recommend Computer System Security— an Overview, pg. 129, as prerequisite reading for this technology description.

Purpose and Origin

Trusted operating systems provide the basic security mechanisms and services that allow a computer system to protect, distinguish, and separate classified data. Trusted operating systems have been developed since the early 1980s and began to receive National Security Agency (NSA) evaluation in 1984.

Technical Detail

Trusted operating systems lower the security risk of implementing a system that processes classified data. Trusted operating systems implement security policies and accountability mechanisms in an operating system package. A security policy is the rules and practices that determine how sensitive information is managed, protected, and distributed [Abrams 95]. Accountability mechanisms are the means of identifying and tracing who has had access to what data on the system so they can be held accountable for their actions.

Trusted operating systems are evaluated by the NSA National Computer Security Center (NCSC) against a series of six requirements-level classes listed in the table below. C1 systems have basic capabilities. A1 systems provide the most capability. The higher the rating level is, the wider the range of classified data is that may be processed.

The table below shows the NCSC Evaluation Criteria Classes.

Class	Title	Number of Approved Operating Systems in this Class [TPEP 96]
A1	Verified Design	0
B3	Security Domains	1
B2	Structured Protection	1
B1	Labeled Security Protection	7
C2	Controlled Access Protection	5
C1	Discretionary Security Protection	No Longer Evaluated

A low level (C1 and C2) system provides limited discretionary access controls and identification and authentication mechanisms. Discretionary access controls identify who can have access to system data based on the need to know. Mandatory access controls identify who or what process can have access to data based on the requester having formal

clearance for the security level of the data. A low-level system is used when the system only needs to be protected against human error and it is unlikely that a malicious user can gain access to the system.

A higher level (B2, B3, and A1) system provides complete mandatory and discretionary access control, thorough security identification of data devices, rigid control of transfer of data and access to devices, and complete auditing of access to the system and data. These higher level systems are used when the system must be protected against a malicious user's abuse of authority, direct probing, and human error [Abrams 95].

The portion of the trusted operating system that grants requesters access to data and records the action is frequently called the reference monitor because it refers to an authorization database to determine if access should be granted. Higher level trusted operating systems are used in MLS hosts and compartmented mode workstations (see Computer System Security— an Overview, pg. 129, for overview information).

Usage Considerations

Trusted operating systems must be used to implement multi-level security systems and to build security guards that allow systems of different security levels to be connected to exchange data. Use of a trusted operating system may be the only way that a system can be networked with other high security systems. Trusted operating systems may be required if a C4I system processes intelligence data and provides data to war fighters. Department of Defense (DoD) security regulations define what evaluation criteria must be satisfied for a multi-level system based on the lowest and highest classification of the data in a system and the clearance level of the users of the system. Using an NCSC-evaluated system reduces accreditation cost and risk. The security officer identified as the Designated Approving Authority (DAA) for secure computer systems has the responsibility and authority to review and approve the systems to process classified information. The DAA will require analysis and tests of the system to assure that it will operate securely. The DAA can accept the NCSC evaluation of a system rather than generating the data. For a B3 or A1 system, that can represent a savings of 1 to 2 years in schedule and the operating system will provide a proven set of functions.

Maturity

This technology has been implemented by several vendors for commercial-off-the-shelf (COTS) use in secure systems. As of September 1996, the NCSC Evaluated Product List indicated that fourteen operating systems have been evaluated as level C2, B1,B2, and B3 systems in the last three years [TPEP 96]. The number of operating systems evaluated by class (excluding evaluations of updated versions of operating systems) is included in the table on page 377. Use of one of the approved trusted

operating systems can result in substantial cost and schedule reductions for a system development effort and provide assurance that the system can be operated securely.

Costs and Limitations

The heavy access control and accounting associated with high security systems can affect system performance; as such, higher performance processors, I/O, and interfaces may be required. Trusted operating systems have unique interfaces and operating controls that require special security knowledge to use and operate. Frequently COTS products that operate satisfactorily with a standard operating system must be replaced or augmented to operate with a trusted operating system.

Dependencies

Trusted operating systems at B2 and above enable the development of system interoperability for systems at different security levels and allow applications to perform data fusion. They are dependent on a trusted computing base that provides secure data paths and protected memory.

Index Categories

Name of technology	Trusted Operating Systems
Application category	Trusted Operating Systems (AP.2.4.1)
Quality measures category	Security (QM.2.1.5)
Computing reviews category	Operating System Security and Protection (D.4.6), Computer-Communications Network Security Protection (C.2.0)

References and Information Sources

[Abrams 95] Abrams, Marshall D.; Jajodia, Sushil; & Podell, Harold J. *Information Security An Integrated Collection of Essays.* Los Alamitos, CA: IEEE Computer Society Press, 1995.

[Russel 91] Russel, Deborah & Gangemi, G.T. Sr. *Computer Security Basics.* Sebastopol, CA: O'Reilly & Associates, 1991.

[TPEP 96] *Trusted Product Evaluation Program Evaluated Product List* [online]. Available WWW <URL: http://www.radium.ncsc.mil/tpep/index.html> (1996).

[White 96] White, Gregory B.; Fisch, Eric A.; & Pooch, Udo W. *Computer System and Network Security.* Boca Raton, FL: CRC Press, 1996.

Author

Tom Mills, Loral
TMILLS@ccs.lmco.com

Last Modified

10 Jan 97

Two Tier Software Architectures

COMPLETE

Note

We recommend Client/Server Software Architectures, pg. 101, as prerequisite reading for this technology description.

Purpose and Origin

Two tier software architectures were developed in the 1980s from the file server software architecture design. The two tier architecture is intended to improve *usability* by supporting a forms-based, user-friendly interface. The two tier architecture improves *scalability* by accommodating up to 100 users (file server architectures only accommodate a dozen users), and improves *flexibility* by allowing data to be shared, usually within a homogeneous environment [Schussel 96]. The two tier architecture requires minimal operator intervention, and is frequently used in non-complex, non-time critical information processing systems. Detailed readings on two tier architectures can be found in Schussel and Edelstein [Schussel 96, Edelstein 94].

Technical Detail

Two tier architectures consist of three components distributed in two layers: client (requester of services) and server (provider of services). The three components are

1. User System Interface (such as session, text input, dialog, and display management services)
2. Processing Management (such as process development, process enactment, process monitoring, and process resource services)
3. Database Management (such as data and file services)

The two tier design allocates the user system interface exclusively to the client. It places database management on the server and splits the processing management between client and server, creating two layers. Figure 30 depicts the two tier software architecture.

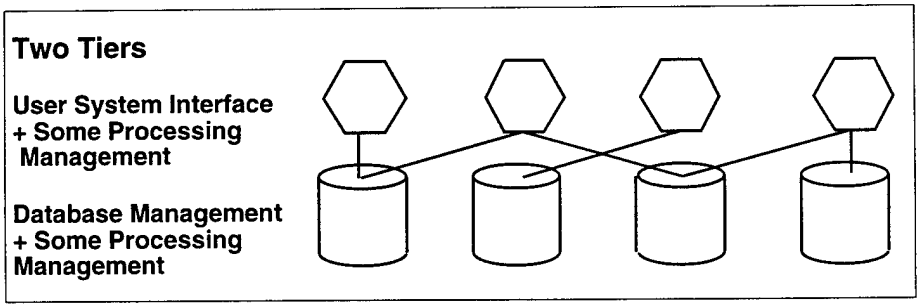


Figure 30: Two Tier Client Server Architecture Design [Louis 95]

In general, the user system interface client invokes services from the database management server. In many two tier designs, most of the appli-

cation portion of processing is in the client environment. The database management server usually provides the portion of the processing related to accessing data (often implemented in store procedures). Clients commonly communicate with the server through SQL statements or a call-level interface. It should be noted that connectivity between tiers can be dynamically changed depending upon the user's request for data and services.

As compared to the file server software architecture (that also supports distributed systems), the two tier architecture improves flexibility and scalability by allocating the two tiers over the computer network. The two tier improves usability (compared to the file sever software architecture) because it makes it easier to provide a customized user system interface.

It is possible for a server to function as a client to a different server— in a hierarchical client/server architecture. This is known as a chained two tier architecture design.

Usage Considerations

Two tier software architectures are used extensively in non-time critical information processing where management and operations of the system are not complex. This design is used frequently in decision support systems where the transaction load is light. Two tier software architectures require minimal operator intervention. The two tier architecture works well in relatively homogeneous environments with processing rules (business rules) that do not change very often and when workgroup size is expected to be fewer than 100 users, such as in small businesses.

Maturity

Two tier client/server architectures have been built and fielded since the middle to late 1980s. The design is well known and used throughout industry. Two tier architecture development was enhanced by fourth generation languages.

Costs and Limitations

Scalability. The two tier design will scale-up to service 100 users on a network. It appears that beyond this number of users, the performance capacity is exceeded. This is because the client and server exchange "keep alive" messages continuously, even when no work is being done, thereby saturating the network [Schussel 96].

Implementing business logic in stored procedures can limit scalability because as more application logic is moved to the database management server, the need for processing power grows. Each client uses the server to execute some part of its application code, and this will ultimately reduce the number of users that can be accommodated.

Interoperability. The two tier architecture limits interoperability by using stored procedures to implement complex processing logic (such as managing distributed database integrity) because stored procedures are normally implemented using a commercial database management system's proprietary language. This means that to change or interoperate with more than one type of database management system, applications may need to be rewritten. Moreover, database management system's proprietary languages are generally not as capable as standard programming languages in that they do not provide a robust programming environment with testing and debugging, version control, and library management capabilities.

System administration and configuration. Two tier architectures can be difficult to administer and maintain because when applications reside on the client, every upgrade must be delivered, installed, and tested on each client. The typical lack of uniformity in the client configurations and lack of control over subsequent configuration changes increase administrative workload.

Batch jobs. The two tiered architecture is not effective running batch programs. The client is typically tied up until the batch job finishes, even if the job executes on the server; thus, the batch job and client users are negatively affected [Edelstein 94].

Dependencies

Developing a two tier client/server architecture following an object-oriented methodology would be dependent on the CORBA standards for design implementation. See Common Object Request Broker Architecture, pg. 107.

Alternatives

Possible alternatives for two tier client server architectures are

- the three-tier architecture (see pg. 367) if there is a requirement to accommodate greater than 100 users
- distributed/collaborative architectures (see pg. 163) if there is a requirement to design on an enterprise-wide scale. An enterprise-wide design is comprised of numerous smaller systems or subsystems.

When preparing a two tier architecture for possible migration to an alternative three tier architecture, the following five steps will make the transition less costly and of lower risk [Dickman 95]:

1. Eliminate application diversity by ensuring a common, cross-hardware library and development tools.
2. Develop smaller, more comparable service elements, and allow access through clearly-defined interfaces.

3. Use an Interface Definition Language (IDL) to model service interfaces and build applications using header files generated when compiled.
4. Place service elements into separate directories or files in the source code.
5. Increase flexibility in distributed functionality by inserting service elements into Dynamic Linked Libraries (DLLs) so that they do not need to be complied into programs.

Complementary Technologies

Complementary technologies for two tier architectures are CASE (computer-aided software engineering) tools because they facilitate two tier architecture development, and open systems (see pg. 135) because they facilitate developing architectures that improve scalability and flexibility.

Index Categories

Name of technology	Two Tier Software Architectures
Application category	Client/Server (AP.2.1.2.1)
Quality measures category	Usability (QM.2.3), Maintainability (QM.3.1), Scalability (QM.4.3)
Computing reviews category	Distributed Systems (C.2.4), Software Engineering Design (D.2.10)

References and Information Sources

[Dickman 95] Dickman, A. "Two-Tier Versus Three-Tier Apps." *Informationweek* 553 (November 1995): 74-80.

[Edelstein 94] Edelstein, Herb. "Unraveling Client Server Architectures." *DBMS 7,* 5 (May 1994): 34(7).

[Gallaugher 96] Gallaugher, J. & Ramanathan, S. "Choosing a Client/Server Architecture. A Comparison of Two-Tier and Three-Tier Systems." *Information Systems Management Magazine 13,* 2 (Spring 1996): 7-13.

[Louis 95] *Louis* [online]. Available WWW <URL: http://www.softis.is > (1995).

[Newell 95] Newell, D.; Jones, O.; & Machura, M. "Interoperable Object Models for Large Scale Distributed Systems," 152+32. *Proceedings. International Seminar on Client/Server Computing.* La Hulpe, Belgium, October 30-31, 1995. London, UK: IEE, 1995.

[Schussel 96] Schussel, George. *Client/Server: Past, Present and Future* [online]. Available WWW <URL: http://www.dciexpo.com/geos/> (1996).

Author

Darleen Sadoski, GTE
sadoski.darleen@mail.ndhm.gtegsc.com

External Reviewer(s)	Paul Clements, SEI Frank Rogers, GTE
Last Modified	10 Jan 97

Virus Detection

ADVANCED

Note

We recommend Computer System Security— an Overview, pg. 129, as prerequisite reading for this technology description.

Purpose and Origin

Technologies for Computer System Security in C4I Systems (see pg. 129) introduced virus detection software as one of the system security mechanisms included in Intranets used to support C4I systems. Viruses are malicious segments of code, inserted into legitimate programs, that execute when the legitimate program is executed. The primary characteristic of a virus is that it replicates itself when it is executed and inserts the replica into another program which will replicate the virus again when it executes. A computer is said to be infected if it contains a virus. Detecting that a computer is infected is the process of virus detection. Viruses have existed since the early 1980s and programs to detect them have been developed since then [Denning 90].

Technical Detail

Since viruses are executable code, they are written for a particular processor. They have been written for mainframes, for UNIX machines, and for personal computers (IBM PC compatibles and Apple Macintoshes). By far the most viruses have been developed to attack 80x86-based IBM PC compatible computers. By 1996, there have been over 2000 kinds of viruses developed that attack IBM PC compatible computers. The IBM PC compatible is a frequent target of viruses because there are so many of that type of computer in use and the operating system (DOS and Windows) has no provision to prevent code from being modified. A few viruses, written using word processing or spreadsheet macros, infect any processor that runs the word processor or spreadsheet program that can interpret those macros. There were some early, much publicized, viruses on UNIX machines, but they are rare. The 1988 Morris Worm was an early example of malicious code that attacked UNIX machines [Spafford 88]. Viruses are hard to write because they require detailed knowledge of how the operating system works; there are much easier ways to damage or copy information on a UNIX computer. There have been a few mainframe viruses but they are also rare because mainframe operating systems make it difficult for a program to gain access to and modify other programs.

Within some viruses is a portion of code called the payload. The payload is designed to do something malicious such as corrupt files, display a message on the screen, or prevent the computer from booting. When the virus executes or at some future execution after a trigger condition has been met, the virus will execute the payload. A favorite trigger condition is the occurrence of a particular date, such as Friday the 13th. A virus still

causes harm, even if it does not contain a payload, by consuming processor and storage resources as it replicates itself.

The two general types of PC viruses are boot-record infectors and program file infectors. The type is determined by where the virus code copy is written when it is replicated.

Boot-record infectors, also called system infectors, infect the boot records on hard disks and floppy disks. When the system is booted, they are loaded into memory. They may execute and replicate themselves every time a disk is loaded. Once a hard disk boot record is infected the virus will be loaded into memory each time the system is booted from the hard disk.

The program file infectors attach their replicas to program file (.EXE or .COM files) hosts on disk whenever the virus is executed. When the host is executed the virus replicates itself again. When the virus is added to a file it makes the file larger. In order to not cause an obvious growth in a file, viruses include a signature pattern in the copy that it can recognize so that it will not add to a file again if the virus is there already.

There are three basic types of virus detection software:

- virus scanner
- activity monitor
- change detection

Virus scanner software looks for the virus signature in memory or in program files and looks for code in the boot record that is not boot code. Once suspicious code is found, a message is displayed to the operator that the system is infected. Some virus scanners have the capability to remove viruses as well as to detect them.

Activity monitors are memory resident programs that watch for suspicious activity such as a program other than the operating system trying to format disks, delete an executable file, or change the file allocation table on a disk. They also may look for programs trying to go memory resident, scanning for other program files, or trying to modify their own code [Slade 96].

Change detection software scans the executable program files in the system before a system is used and records vital statistics about each program, such as program file length or a calculated CRC or checksum. After the system is in operation, the change detection software periodi-

cally scans the program files looking for changes compared to the prestored data. These changes could have been caused by a virus.

Usage Considerations

Virus scanners are executed periodically, when the system is started up, or whenever a disk is initially put into the system. When new software (commercial, freeware, or downloaded) is added to the system, it should be checked with a virus scanner before the new software is executed to identify known viruses if they are present. Although virus scanners are very useful in finding known viruses they will not detect new kinds of viruses. They therefore must be updated frequently to include the "signatures" of new viruses.

Activity monitors are more likely to find new types of viruses than virus scanners since activity monitors are not limited to finding a known bit pattern in memory or on disk. Activity monitors have considerable performance overhead since they must be constantly scanning for unusual activity. Activity monitors also must be incorporated into software change processes so that its baseline of "correct" software files can be maintained.

Of the three types of virus detection software, change detection software has the best chance of detecting current and future virus types but is most likely to produce false alarms [Slade 96]. The database for change detection software must be updated every time system files or executable program files are updated. This adds maintenance overhead to the system if the system is frequently modified.

Maturity

More than 100 virus detection products are listed on the National Institute of Standards and Technology (NIST) list of products reviewed [NIST 96]. Most of those products are virus scanners. Virus scanners are also the most rapidly changing as they must be updated to check for new virus "signatures" as new viruses are identified. The challenge to virus detection product vendors is in the constant race to keep up with the host of smart computer hackers and malicious software developers creating new strains of viruses.

Costs and Limitations

Effective use of virus detection software requires system administrators familiar with virus types and their mode of attack, the operation of the virus detection software, the ability to evaluate the virus detection program output, and the ability to recognize a true attack versus a false alarm. This requires knowledge of the system and its normal operation, training in the use of the virus detection software, and frequent retraining as the virus detection software is routinely updated.

Index Categories

Name of technology	Virus Detection
Application category	Information Security (AP.2.4)
Quality measures category	Security (QM.2.1.5), Denial of Service (QM.2.1.4.1.3)
Computing reviews category	Operating Systems Security and Protection (D.4.6), Security and Protection (K.6.5)

References and Information Sources

[Abrams 95] Abrams, Marshall D.; Jajodia, Sushil; & Podell, Harold J. *Information Security An Integrated Collection of Essays*. Los Alamitos, CA: IEEE Computer Society Press, 1995.

[Denning 90] Denning, Peter J. *Computers Under Attack Intruders, Worms and Viruses*. New York, NY: ACM Press, 1990.

[Garfinkel 96] Garfinkel, Simson & Spafford, Gene. *Practical UNIX and Internet Security* Second Edition. Sebastopol, CA: O'Reilly & Associates, 1996.

[NIST 96] *Virus Information* product review [online]. National Institute of Standards and Technology (NIST) Computer Security Resource Clearinghouse. Available WWW <URL: http://csrc.ncsl.nist.gov/virus/virusrevws/> (1996).

[Russel 91] Russel, Deborah & Gangemi, G.T., Sr. *Computer Security Basics*. Sebastopol, CA: O'Reilly & Associates, 1991.

[Slade 96] Slade, Robert. *Reviewing Anti-virus Products* [online]. Available WWW <URL: http://csrs.ncsl.nist.gov/virus/guidance.sla> (1996).

[Spafford 88] Spafford, Eugene H. *The Internet Worm: An Analysis* (CSD-TR-823). West Lafayette, IN: Purdue University, 1988.

Author Tom Mills, Loral
TMILLS@ccs.lmco.com

Last Modified 10 Jan 97

References

[ARC 96] Laforme, Deborah & Stropky, Maria E. *An Automated Mechanism for Effectively Applying Domain Engineering in Reuse Activities* [online]. Available WWW <URL: http://arc_www2.belvoir.army.mil/htmldocs/arc/da_papers/applying_domain_engineering.htm> (1996).

[Barbacci 95] Barbacci, Mario; Klein, Mark H.; Longstaff, Thomas H. & Weinstock, Charles B. *Quality Attributes* (CMU/SEI-95-TR-021). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1995.

[Boehm 78] Boehm, Barry W.; Brown, John R.; Kaspar, Hans; Lipow, Myron; MacLeod, Gordon J. & Merritt, Michael J. *Characteristics of Software Quality.* New York, NY: North-Holland Publishing Company, 1978.

[Clements 96] Clements, Paul C. & Northrop, Linda M. *Software Architecture: An Executive Overview* (CMU/SEI-96-TR-003). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1996.

[Deutsch 88] Deutsch, Michael S. & Willis, Ronald R. *Software Quality Engineering: A Total Technical and Management Approach.* Englewood Cliffs, NJ: Prentice-Hall, 1988.

[DoD 91] Department of Defense. *Software Technology Strategy.* DRAFT: December, 1991.

[Evans 87] Evans, Michael W. & Marciniak, John. *Software Quality Assurance and Management.* New York, NY: John Wiley & Sons, Inc., 1987.

[Gotel 95] Gotel, Orlena. *Contribution Structures for Requirements Traceability.* London, England: Imperial College, Department of Computing, 1995.

[IEEE 87] IEEE Std 1002-1987. *IEEE Standard Taxonomy for Software Engineering Standards*. New York, NY: Institute of Electrical and Electronics Engineers, 1987.

[IEEE 90] Institute of Electrical and Electronics Engineers. *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*, New York, NY: 1990.

[IEEE 91] IEEE Std 1074-1991. *IEEE Standard for Developing Life Cycle Processes*. New York, NY: Institute of Electrical and Electronics Engineers, 1991.

[ITS 96] *Letter-by-Letter Listing* [online]. Available WWW <URL: http: //www.its.bldrdoc.gov/fs-1037/dir-001/_0064.htm> (1996).

[McDaniel 94] McDaniel, George, ed. *IBM Dictionary of Computing*. New York, NY: McGraw-Hill, Inc., 1994.

[McGill 96] *The Software Agents Mailing List FAQ* [online]. Available WWW <URL:http://www.ee.mcgill.ca/~belmarc/agent_faq.html> (1996).

[Poore 96] Poore, Jesse. *Re: Definition for Statistical Testing* [email to Gary Haines], [online]. Available email: ghaines@spacecom.af.mil (October 2, 1996).

[SEI 96] *What is Model Based Software Engineering (MBSE)?* [online]. Available WWW <URL: http://www.sei.cmu.edu/technology/mbse/is.html> (1996).

[Toronto 95] comp.human-factors faq WWW page [online]. Available WWW <URL: http://www/dgp.toronto.edu/people/ematias/faq/G/G-1.html> (1995).

[Webster 87] *Webster's Ninth New Collegiate Dictionary*. Springfield, MA: Merriam-Webster Inc., 1987.

Glossary

Abstractness — the degree to which a system or component performs only the necessary functions relevant to a particular purpose.

Acceptance testing — formal testing conducted to determine whether or not a system satisfies its acceptance criteria and to enable the customer to determine whether or not to accept the system [IEEE 90].

Accessibility

1. (Denial of Service) the degree to which the software system protects system functions or service from being denied to the user
2. (Reusability) the degree to which a software system or component facilitates the selective use of its components [Boehm 78].

Accountability — the ability of a system to keep track of who or what accessed and/or made changes to the system.

Accuracy — a quantitative measure of the magnitude of error [IEEE 90].

Acquisition cycle time — the period of time that starts when a system is conceived and ends when the product meets its initial operational capability.

Adaptability — the ease with which software satisfies differing system constraints and user needs [Evans 87].

Adaptive maintenance — software maintenance performed to make a computer program usable in a changed environment [IEEE 90].

Adaptive measures — a category of quality measures that address how easily a system can evolve or migrate.

Agent — a piece of software which acts to accomplish tasks on behalf of its user [McGill 96].

Anonymity — the degree to which a software system or component allows for or supports anonymous transactions.

Application program interface — a formalized set of software calls and routines that can be referenced by an application program in order to access supporting system or network services [ITS 96].

Architectural design — the process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system [IEEE 90].

Artificial intelligence — a subfield within computer science concerned with developing technology to enable computers to solve problems (or assist humans in solving

problems) using explicit representations of knowledge and reasoning methods employing that knowledge [DoD 91].

Auditable the degree to which a software system records information concerning transactions performed against the system.

Availability the degree to which a system or component is operational and accessible when required for use [IEEE 90].

Capacity a measure of the amount of work a system can perform [Barbacci 95].

Code the transforming of logic and data from design specifications (design descriptions) into a programming language [IEEE 90].

Commonality the degree to which standards are used to achieve interoperability.

Communication software software concerned with the representation, transfer, interpretation, and processing of data among computer systems or networks. The meaning assigned to the data must be preserved during these operations.

Compactness the degree to which a system or component makes efficient use of its data storage space- occupies a small volume.

Compatibility the ability of two or more systems or components to perform their required functions while sharing the same hardware or software environment [IEEE 90].

Completeness the degree to which all the parts of a software system or component are present and each of its parts is fully specified and developed [Boehm 78].

Complexity

1. (Apparent) the degree to which a system or component has a design or implementation that is difficult to understand and verify [IEEE 90].
2. (Inherent) the degree of complication of a system or system component, determined by such factors as the number and intricacy of interfaces, the number and intricacy of conditional branches, the degree of nesting, and the types of data structures [Evans 87].

Component testing testing of individual hardware or software components or groups of related components [IEEE 90].

Concept phase the initial phase of a software development project, in which the user needs are described and evaluated through documentation (for example, statement of needs, advance planning report, project initiation memo, feasibility studies, system definition, documentation, regulations, procedures, or policies relevant to the project) [IEEE 90].

Conciseness the degree to which a software system or component has no excessive information present.

Confidentiality the nonoccurrence of the unauthorized disclosure of information [Barbacci 95].

Consistency the degree of uniformity, standardization, and freedom from contradiction among the documents or parts of a system or component [IEEE 90].

Corrective maintenance maintenance performed to correct faults in hardware or software [IEEE 90].

Correctness the degree to which a system or component is free from faults in its specification, design, and implementation [IEEE 90].

Cost estimation the process of estimating the "costs" associated with software development projects, to include the effort, time, and labor required.

Cost of maintenance the overall cost of maintaining a computer system to include the costs associated with personnel, training, maintenance control, hardware and software maintenance, and requirements growth.

Cost of operation the overall cost of operating a computer system to include the costs associated with personnel, training, and system operations.

Cost of ownership the overall cost of a computer system to an organization to include the costs associated with operating and maintaining the system, and the lifetime of operational use of the system

Data management the function that provides access to data, performs or monitors the storage of data, and controls input/output operations [McDaniel 94].

Data management security the protection of data from unauthorized (accidental or intentional) modification, destruction, or disclosure [ITS 96].

Data recording to register all or selected activities of a computer system. Can include both external and internal activity.

Data reduction any technique used to transform data from raw data into a more useful form of data. For example, grouping, summing, or averaging related data [IEEE 90].

Database

1. a collection of logically related data stored together in one or more computerized files. Note: Each data item is identified by one or more keys [IEEE 90].
2. an electronic repository of information accessible via a query language interface [DoD 91].

Database administration the responsibility for the definition, operation, protection, performance, and recovery of a database [IEEE 90].

Database design the process of developing a database that will meet a user's requirements. The activity includes three separate but dependent steps: conceptual database design, logical database design, and physical database design [IEEE 91].

Denial of service the degree to which a software system or component prevents the interference or disruption of system services to the user.

Dependability that property of a computer system such that reliance can justifiably be placed on the service it delivers [Barbacci 95].

Design phase the period of time in the software life cycle during which the designs for architecture, software components, interfaces, and data are created, documented, and verified to satisfy requirements [IEEE 90].

Detailed design the process of refining and expanding the preliminary design of a system or component to the extent that the design is sufficiently complete to be implemented [IEEE 90].

Distributed computing a computer system in which several interconnected computers share the computing tasks assigned to the system [IEEE 90].

Domain analysis the activity that determines the common requirements within a domain for the purpose of identifying reuse opportunities among the systems in the domain. It builds a domain architectural model representing the commonalities and differences in requirements within the domain (problem space) [ARC 96].

Domain design the activity that takes the results of domain analysis to identify and generalize solutions for those common requirements in the form of a Domain-Specific Software Architecture (DSSA). It focuses on the problem space, not just on a particular system's requirements, to design a solution (solution space) [ARC 96].

Domain engineering the process of analysis, specification and implementation of software assets in a domain which are used in the development of multiple software products [SEI 96]. The three main activities of domain engineering are: domain analysis, domain design, and domain implementation [ARC 96].

Domain implementation the activity that realizes the reuse opportunities identified during domain analysis and design in the form of common requirements and design solutions, respectively. It facilitates the integration of those reusable assets into a particular application [ARC 96].

Effectiveness the degree to which a system's features and capabilities meet the user's needs.

Efficiency the degree to which a system or component performs its designated functions with minimum consumption of resources (CPU, Memory, I/O, Peripherals, Networks) [IEEE 90].

Error handling the function of a computer system or component that identifies and responds to user or system errors to maintain normal or at the very least degraded operations.

Error proneness the degree to which a system may allow the user to intentionally or unintentionally introduce errors into or misuse the system.

Error tolerance the ability of a system or component to continue normal operation despite the presence of erroneous inputs [IEEE 90].

Evolvability the ease with which a system or component can be modified to take advantage of new software or hardware technologies.

Expandability see Extendability [IEEE 90].

Extendability the ease with which a system or component can be modified to increase its storage or functional capacity [IEEE 90].

Fail safe pertaining to a system or component that automatically places itself in a safe operating mode in the event of a failure [IEEE 90].

Fail soft pertaining to a system or component that continues to provide partial operational capability in the event of certain failures [IEEE 90].

Fault an incorrect step, process, or data definition in a computer program [IEEE 90].

Fault tolerance the ability of a system or component to continue normal operation despite the presence of hardware or software faults [IEEE 90].

Fidelity the degree of similarity between a model and the system properties being modeled [IEEE 90].

Flexibility the ease with which a system or component can be modified for use in applications or environments other than those for which it was specifically designed [IEEE 90].

Functional scope the range or scope to which a system component is capable of being applied.

Functional testing testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions. Synonym: black-box testing [IEEE 90].

Generality the degree to which a system or component performs a broad range of functions [IEEE 90].

Graphics methods and techniques for converting data to or from graphic display via computers [McDaniel 94].

Hardware maintenance the cost associated with the process of retaining a hardware system or component in, or restoring it to, a state in which it can perform its required functions.

Human Computer Interaction a subfield within computer science concerned with the design, evaluation, and implementation of interactive computing systems for human use and with the study of major phenomena surrounding them [Toronto 95].

Human engineering the extent to which a software product fulfills its purpose without wasting user's time and energy or degrading their morale [Boehm 78].

Implementation phase the period of time in the software life cycle during which a software product is created from design documentation and debugged [IEEE 90].

Incompleteness the degree to which all the parts of a software system or component are not present and each of its parts is not fully specified or developed.

Information Security the concepts, techniques, technical measures, and administrative measures used to protect information assets from deliberate or inadvertent unauthorized acquisition, damage, disclosure, manipulation, modification, loss, or use [McDaniel 94].

Installation and checkout phase the period of time in the software life cycle during which a software product is integrated into its operational environment and tested in this environment to ensure it performs as required [IEEE 90].

Integration testing testing in which software components, hardware components, or both are combined and tested to evaluate the interaction between them [IEEE 90].

Integrity the degree to which a system or component prevents unauthorized access to, or modification of, computer programs or data [IEEE 90].

Interfaces design the activity concerned with the interfaces of the software system contained in the software requirements and software interface requirements documentation. Consolidates the interface descriptions into a single interface description of the software system [IEEE 91].

Interface testing testing conducted to evaluate whether systems or components pass data and control correctly to one another [IEEE 90].

Interoperability the ability of two or more systems or components to exchange information and to use the information that has been exchanged [IEEE 90].

Latency the length of time it takes to respond to an event [Barbacci 95].

Lifetime of operational capability the total period of time in a system's life that it is operational and meeting the user's needs.

Maintainability The ease with which a software system or component can be modified to correct faults, improve performance, or other attributes, or adapt to a changed environment [IEEE 90].

Maintenance control the cost of planning and scheduling hardware preventive maintenance, and software maintenance and upgrades, managing the hardware and software baselines, and providing response for hardware corrective maintenance.

Maintenance measures a category of quality measures that address how easily a system can be repaired or changed.

Maintenance personnel the number of personnel needed to maintain all aspects of a computer system, including the support personnel and facilities needed to support that activity.

Manufacturing phase the period of time in the software life cycle during which the basic version of a software product is adapted to a specified set of operational environments and is distributed to a customer base [IEEE 90].

Model an approximation, representation, or idealization of selected aspects of the structure, behavior, operation, or other characteristics of a real-world process, concept, or system. Note: Models may have other models as components [IEEE 90].

Modifiability the degree to which a system or component facilitates the incorporation of changes, once the nature of the desired change has been determined [Boehm 78].

Necessity of characteristics the degree to which all of the necessary features and capabilities are present in the software system

Need satisfaction measures a category of quality measures that address how well a system meets the user's needs and requirements.

Network management the execution of the set of functions required for controlling, planning, allocating, deploying, coordinating, and monitoring the resources of a computer network [ITS 96].

Openness the degree to which a system or component complies with standards.

Operability the ease of operating the software [Deutsch 88].

Operational testing testing conducted to evaluate a system or component in its operational environment [IEEE 90].

Operations and maintenance phase the period of time in the software life cycle during which a software product is employed in its operational environment, monitored for satisfactory performance, and modified as necessary to correct problems or to respond to changing requirements [IEEE 90].

Operations personnel the number of personnel needed to operate all aspects of a computer system, including the support personnel and facilities needed to support that activity.

Operations system the cost of environmentals, communication, licenses, expendables, and documentation maintenance for an operational system.

Organizational measures a category of quality measures that address how costly a system is to operate and maintain.

Parallel computing a computer system in which interconnected processors perform concurrent or simultaneous execution of two or more processes [McDaniel 94].

Perfective maintenance software maintenance performed to improve the performance, maintainability, or other attributes of a computer program [IEEE 90].

Performance measures a category of quality measures that address how well a system functions.

Performance testing testing conducted to evaluate the compliance of a system or component with specified performance requirements [IEEE 90].

Portability The ease with which a system or component can be transferred from one hardware or software environment to another [IEEE 90].

Productivity the quality or state of being productive [Webster 87].

Protocol a set of conventions that govern the interaction of processes, devices, and other components within a system [IEEE 90].

Provably correct the ability to mathematically verify the correctness of a system or component.

Qualification phase the period of time in the software life cycle during which it is determined whether a system or component is suitable for operational use.

Qualification testing testing conducted to determine whether a system or component is suitable for operational use [IEEE 90].

Quality measure a software feature or characteristic used to assess the quality of a system or component.

Readability the degree to which a system's functions and those of its component statements can be easily discerned by reading the associated source code.

Real-time responsiveness the ability of a system or component to respond to an inquiry or demand within a prescribed time frame.

Recovery the restoration of a system, program, database, or other system resource to a prior state following a failure or externally caused disaster; for example, the restoration of a database to a point at which processing can be resumed following a system failure [IEEE 90].

Reengineering rebuilding a software system or component to suit some new purpose; for example to work on a different platform, to switch to another language, to make it more maintainable.

Regression testing selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements [IEEE 90].

Reliability the ability of a system or component to perform its required functions under stated conditions for a specified period of time [IEEE 90].

Requirements engineering involves all life-cycle activities devoted to identification of user requirements, analysis of the requirements to derive additional requirements, documentation of the requirements as a specification, and validation of the documented requirements against user needs, as well as processes that support these activities [DoD 91].

Requirements growth the rate at which the requirements change for an operational system. The rate can be positive or negative.

Requirements phase the period of time in the software life cycle during which the requirements for a software product are defined and documented [IEEE 90].

Requirements tracing describing and following the life of a requirement in both forwards and backwards direction (i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through periods of ongoing refinement and iteration in any of these phases) [Gotel 95].

Resource utilization the percentage of time a resource (CPU, Memory, I/O, Peripheral, Network) is busy [Barbacci 95].

Responsiveness the degree to which a software system or component has incorporated the user's requirements.

Restart to cause a computer program to resume execution after a failure, using status and results recorded at a checkpoint [IEEE 90].

Retirement phase the period of time in the software life cycle during which support for a software product is terminated [IEEE 90].

Reusability the degree to which a software module or other work product can be used in more than one computing program or software system [IEEE 90].

Reverse engineering the process of analyzing a system's code, documentation, and behavior to identify its current components and their dependencies to extract and create system abstractions and design information. The subject system is not altered; however, additional knowledge about the system is produced.

Robustness the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environment conditions [IEEE 90].

Safety a measure of the absence of unsafe software conditions. The absence of catastrophic consequences to the environment [Barbacci 95].

Scalability the ease with which a system or component can be modified to fit the problem area.

Security the ability of a system to manage, protect, and distribute sensitive information.

Select or develop algorithms the activity concerned with selecting or developing a procedural representation of the functions in the software requirements documentation for each software component and data structure. The algorithms shall completely satisfy the applicable functional and/or mathematical specifications [IEEE 91].

Self-descriptiveness the degree to which a system or component contains enough information to explain its objectives and properties. [IEEE 90].

Simplicity the degree to which a system or component has a design and implementation that is straightforward and easy to understand [IEEE 90].

Software architecture

the structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time [Clements 96].

Software change cycle time

the period of time that starts when a new system requirement is identified and ends when the requirement has been incorporated into the system and delivered for operational use

Software life cycle

the period of time that begins when a software product is conceived and ends when the software is no longer available for use. The life cycle typically includes a concept phase, requirements phase, design phase, implementation phase, test phase, installation and checkout phase, operation and maintenance phase, and sometimes, retirement phase. These phases may overlap or be performed iteratively, depending on the software development approach used [IEEE 90].

Software maintenance

the cost associated with modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment.

Software migration and evolution

see Adaptive maintenance.

Software upgrade and technology insertion

see Perfective maintenance.

Speed

the rate at which a software system or component performs its functions.

Statistical testing

employing statistical science to evaluate a system or component. Used to demonstrate a system's fitness for use, to predict the reliability of a system in an operational environment, to efficiently allocate testing resources, to predict the amount of testing required after a system change, to qualify components for reuse, and to identify when enough testing has been accomplished [Poore 96].

Structural testing

testing that takes into account the internal mechanism of a system or component. Types include branch testing, path testing, statement testing. Synonym: white-box testing [IEEE 90].

Structuredness

the degree to which a system or component possesses a definite pattern of organization of its interdependent parts [Boehm 78].

Sufficiency of characteristics

the degree to which the features and capabilities of a software system adequately meet the user's needs.

Survivability the degree to which essential functions are still available even though some part of the system is down [Deutsch 88].

System allocation mapping the required functions to software and hardware. This activity is the bridge between concept exploration and the definition of software requirements [IEEE 91].

System analysis and optimization a systematic investigation of a real or planned system to determine the information requirements and processes of the system and how these relate to each other and to any other system, and to make improvements to the system where possible.

System security a system function that restricts the use of objects to certain users [McDaniel 94].

System testing testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements [IEEE 90].

Taxonomy a scheme that partitions a body of knowledge and defines the relationships among the pieces. It is used for classifying and understanding the body of knowledge [IEEE 90].

Test an activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component [IEEE 90].

Test drivers software modules used to invoke a module(s) under test and, often, provide test inputs, control and monitor execution, and report test results [IEEE 90].

Test phase the period of time in the software life cycle during which the components of a software product are evaluated and integrated, and the software product is evaluated to determine whether or not requirements have been satisfied [IEEE 90].

Test tools computer programs used in the testing of a system, a component of the system, or its documentation. Examples include monitor, test case generator, timing analyzer [IEEE 90].

Testability the degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met [IEEE 90]. Note: Not only is testability a measurement for software, it can also apply to the testing scheme.

Testing the process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component [IEEE 90].

Throughput the amount of work that can be performed by a computer system or component in a given period of time [IEEE 90].

Traceability the degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another [IEEE 90].

Training Provisions to learn how to develop, maintain, or use the software system.

Trouble report analysis the methodical investigation of a reported operational system deficiency to determine what, if any, corrective action needs to be taken.

Trustworthiness the degree to which a system or component avoids compromising, corrupting, or delaying sensitive information.

Understandability the degree to which the purpose of the system or component is clear to the evaluator [Boehm 78].

Unit testing testing of individual hardware or software units or groups of related units [IEEE 90].

Upgradeability see Evolvability.

Usability the ease with which a user can learn to operate, prepare inputs for, and interpret outputs of a system or component [IEEE 90].

User interface an interface that enables information to be passed between a human user and hardware or software components of a computer system [IEEE 90].

Verifiability the relative effort to verify the specified software operation and performance [Evans 87].

Vulnerability the degree to which a software system or component is open to unauthorized access, change, or disclosure of information and is susceptible to interference or disruption of system services.

Appendix A Submitting Information for Subsequent Editions

The goal of the project team is that this document be current, and continually reflect the latest information. However, with all the technologies that exist, and all the circumstances in which they can be applied, this is a daunting task, requiring significant resources. The situation is further exacerbated because many of the technology areas we have written about are state-of-the-practice technologies which are in very high demand and with great visibility to the technical community— many are also changing at breakneck speed. Potentially, some technology descriptions may be outdated in a few months.

The readers/users of this document can play a significant role in keeping the document current (and expanding it) by

1. Submitting complete descriptions/writeups of technologies not currently in the document (see pages 45-50 for guidelines for submissions). Note: we are *not* looking for submissions that could be construed as advocacy pieces; please provide supporting documentation.
2. Providing additional information about technologies already included in the document. For example, we are interested in expanding the experience base and knowing about results, both good and bad, of using the various technologies, particularly if there are citable references available. Another example of additional information would be new releases of actual and defacto industry standards.
3. Providing critiques pointing out where information in a technology description may be incorrect.

Please send all email submissions to: *str@SEI.CMU.EDU*

Please send surface mail submissions to:

Software Technology Review
c/o Robert Rosenstein
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Appendix B User Feedback

The project team is also interested in knowing how the document helped you accomplish your job (similarly, if the document was not helpful, tell us why, along with suggestions for improvement). If you feel that the document helped you, we would appreciate the following information:

- In what context were you using the document (please include names of projects/programs/efforts and type of application domain; these names will be kept confidential)?
- What section helped you (please be specific)?
- Precisely how did the document help you?
- What is your job title (maintainer, user, developer, acquirer, etc.)?

Please send all email submissions to: *str@SEI.CMU.EDU*

Please send surface mail submissions to:

Software Technology Review
c/o Robert Rosenstein
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Appendix C Scenarios of Use

Scenario Development

We developed a set of exemplar questions that we feel our target audiences ask or get asked when in the process of selecting software technologies or planning technology insertions. To each question, we applied a template which helped us analyze and develop a scenario. These scenarios give you, the reader, a demonstration of the document's utility. The template consists of the following questions:

1. Who (i.e, which audience category) would ask this question?
2. Why would they ask this question?
3. How do you use this document to answer or address the question?
4. What further analysis may be required?

The following section contains the scenarios that we developed. The types of scenarios can be defined as:

1. Software technology specific with an acquisition flavor.
2. Quality measures or Application taxonomy category specific.
3. Long-term, system evolution specific with incorporation of a new software concept.

Scenarios

1. The Air Force acquisition community distributed a Request for Proposal (RFP) for a new mobile command center that could perform integrated theater and strategic missile defense. For the software development portion of the proposal, a defense contractor plans to use the Cleanroom Software Engineering process. What is Cleanroom?

Who would ask: Systems Program Manager

Why would they ask: They have not heard of this development process, so they need to get more information about it in order to properly evaluate the proposal.

How to use this document: The document contains a technology description on Cleanroom Software Engineering. The reader can find it by using the Keyword Index. This would be the first place to start.

Further analysis needed: After reading the description, more questions may arise. Certainly "high-quality software with certified reliability" is extremely desirable for the new system, but "What other organizations have used this development process and what kind of problems did they run into, and what do they consider the strengths of the process to be?" The technology description lists several project experiences using Cleanroom. The reader can use the references associated with the examples to help find answers to the questions above. In addition, looking at any of the noted references, especially the "checked" references, may provide leads to more Cleanroom experiences as well as providing more detailed information about Cleanroom.

Another question may be, "How do we get training on Cleanroom, how will training get accomplished for the maintenance organization, and more importantly, how large of an impact or change is this to the maintenance organization and how they currently do business?" The two sections in the Cleanroom description that will help here are "Usage Considerations" and "Costs and Limitations." Unfortunately, the document can not fully address these questions. They are valid questions, asked due to information that the technology description provided, but they are context specific and items that the acquisition organization will have to take into account when evaluating the proposal.

"Are there any alternatives to Cleanroom and are there any other technologies required to support Cleanroom?" may also be asked. This question is addressed under the Dependent, Alternative, and Complementary Technologies sections of the description. No alternative or dependent technologies are mentioned in the description. Using object-oriented (OO) methods with Cleanroom is noted as providing additional benefits. The evaluator of the proposal notices that the contractor did not mention anything about object-oriented methods. The evaluator may want to ask the contractor to address how object-oriented methods will be integrated with the process, or if the contractor decided not to pursue OO, why?

If the acquisition community and the contractor worked from the same knowledge baseline (i.e., they both used and referenced the C4 Software Technology Reference Guide), then

many of the questions above could have been addressed in the proposal. The contractor would have known exactly what the evaluator would use as a technical resource and could address the issues and explain how the contractor planned on addressing those issues. Time in the evaluation process might have been saved.

Even though the document may not give the evaluator the context specific information that is needed, it certainly provides a good basic understanding of Cleanroom and multiple other information resources to pursue.

2. The CMAS (Cheyenne Mountain Air Station) Message Processor (CMP) is coming online in 1998. What can we do to ensure that this system is maintainable for several years?

Who would ask: Developer, Systems Program Manager, Maintainer

Why would they ask: Maintenance is a high-cost problem with the current operational systems. Maintenance for a new system is usually the last thing to be thought of and little advanced planning is accomplished.

How to use this document: Some first questions that arise regarding this issue might be, "What are some of the issues regarding software maintenance, and what technologies are out there that we can incorporate to improve software maintainability?" The first place to begin in the document would be to use the Keyword Index. The reader could look up words such as "maintenance" and "maintainability". As with any software technology subject, review these entries for a technology description that may be an overiew of the topic. In this case, there exists an overview description called "Maintenance of Operational Systems-- An Overview". This description provides a framework for the maintenance and reengineering technologies in the document and provides cross-reference pointers to maintenance related technology descriptions. It is recommended to also check for a "Note" section at the beginning of any technology description that the reader may read. In this case, this description points to some recommended concurrent readings. For the most part, this section usually points to prerequisite readings. Both of the questions above can be addressed by looking at this one description.

Going back to the Keyword Index, the reader could find out on which pages the keywords "maintenance" and "maintainability" appear. Referencing these pages may lead the reader to applicable technology descriptions. In addition, the reader will notice that the "maintainability" is a category in the Quality Measures taxonomy (QM.3.1), and that "maintenance, corrective", "maintenance, adaptive" and "maintenance, perfective" are categories in the Application taxonomy, AP.1.9.3.1, AP.1.9.3.2, and AP.1.9.3.3 respectively.

If the reader turns to either the graphical or the textual-based representation of the Quality Measures taxonomy and looks up "Maintainability" (QM.3.1), "Understandability" (QM.3.2) will be found as a related category, with both being subcategories to "Maintenance Measures". By looking under QM.3.1 and QM.3.2 in the taxonomy-based directory, the reader will find numer-

ous technologies that influence these measures. The same can be done by looking up corrective, adaptive, and perfective maintenance in the Application taxonomy. The reader will notice that these categories exist under the "Operations and Maintenance Phase" under which its five major subcategories (AP.1.9.1 - AP.1.9.5) are all legitimate to look at for relevant technologies. This should get the reader well on the way to addressing both questions.

Further analysis needed: After reading the various technology descriptions, other questions may arise, "Can these technologies be applied to CMP, and what are the costs/benefits/limitations that we can expect from incorporating these technologies?" The two sections in technology descriptions that will help here are "Usage Considerations" and "Costs and Limitations." Also, referencing any of the noted experiences or references at the end of a description will provide further detailed information. Unfortunately, the document can not fully address these questions. They are context specific and items that an organization will have to take into account when addressing the maintenance issue.

Note: The third scenario is still under development. The material here will provide intent as to our direction.

3. How do we migrate Granite Sentry (GS) Final Operational Capability (FOC) to an open systems environment?

Who would ask: Systems Program Manager, Maintainer, User

Why would they ask: A new mandate exists that requires systems to be open. GS FOC can be put on that migration path since it has not yet been delivered.

How to use this document: Some first questions in attacking this issue might be, "What does open systems mean, what are the risks involved, and how do we get there?" The document has a technology description on COTS and Open Systems which specifically addresses the first 2 questions stated above. The reader can find it by using the Keyword Index. The third question is context specific to the GS system and the Cheyenne Mountain community. As such, the document can not prescribe a specific path. But where possible, the reader will find references in the description to experiences in migrating to open systems.

Further analysis needed: Some questions that may arise after reading the initial description might be, "What standards exist, what do these standards apply to, what standard(s) do we follow?" The open systems technology description has pointers to standards that are contained in the document and references to where other standards may be found. This addresses the first question. Then, the reader can follow those pointers and read the technology descriptions on various standards. The technology descriptions will help address the second question. The third question is again context specific to the Cheyenne Mountain community, and most likely the Air Force community.

Another question that may arise is, "What technologies are out there that we can take advantage of in making GS open?" The open systems technology description or even the technology descriptions on standards may have pointers and/or references to other software technolo-

gies. The reader can follow the pointers and look up the references. A better way may be to use the Quality Measures taxonomy. In the taxonomy, the measures that are usually associated with open systems (interoperability, openness, portability, compatibility) can be found under QM.4, Adaptive Measures. By looking under these measures, such as Interoperability (QM.4.1) or Portability (QM.4.2), in the taxonomy-based directory, the reader can find technologies that influence those measures. These technologies may be candidates for the Cheyenne Mountain community to take advantage of.

More may be found to address this issue by following 2 other information paths. First, at the end of each technology description are references to documents or Web pages which will provide the reader with more detailed information. Checkmarks appear by the key references that the reader may want to start with. Second, the reader can examine the Index Categories section towards the end of each description. This section identifies where the particular technology has been indexed into the two taxonomies and where literature concerning the technology has been categorized by the ACM's Computing Classification System.

Even though the document does not prescribe what to do in order to make a system maintainable, it does provide background on what some of the maintenance issues are and provides some possible software technology solutions. Enough information should be here that evaluations and tradeoffs on how to proceed with CMP can be accomplished.

Keyword Index

U

V

W

Z

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for Public Release Distribution Unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CMU/SEI-97-HB-001	5. MONITORING ORGANIZATION REPORT NUMBER(S) n/a
6a. NAME OF PERFORMING ORGANIZATION Software Engineering Institute 6b. OFFICE SYMBOL (if applicable) SEI	7a. NAME OF MONITORING ORGANIZATION SEI Joint Program Office
6c. ADDRESS (city, state, and zip code) Carnegie Mellon University Pittsburgh PA 15213	7b. ADDRESS (city, state, and zip code) HQ ESC/AXS 5 Eglin Street Hanscom AFB, MA 01731-2116
8a. NAME OFFUNDING/SPONSORING ORGANIZATION SEI Joint Program Office 8b. OFFICE SYMBOL (if applicable) ESC/AXS	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F19628-95-C-0003
8c. ADDRESS (city, state, and zip code)) Carnegie Mellon University Pittsburgh PA 15213	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO: 63756E PROJECT NO.: N/A TASK NO: N/A WORK UNIT NO.: N/A
11. TITLE (Include Security Classification) C4 Software Technology Reference Guide— A Prototype	
12. PERSONAL AUTHOR(S) J. Foreman, J. Gross, R. Rosenstein, D. Fisher, K. Brune, et al	
13a. TYPE OF REPORT Final 13b. TIME COVERED FROM TO	14. DATE OF REPORT (year, month, day) January 10, 1997 15. PAGE COUNT 435
16. SUPPLEMENTARY NOTATION	

17. COSATI CODES			18. SUBJECT TERMS (continue on reverse of necessary and identify by block number)
FIELD	GROUP	SUB. GR.	software technology, C4I, software reference, C4I systems, software technology reference guide, taxonomies

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

The Air Force acquisition community tasked the Software Engineering Institute (SEI) to create a reference document that would provide the Air Force with a better understanding of software technologies. This knowledge will allow the Air Force to systematically plan the research and development (R&D) and technology insertion required to meet current and future Air Force needs, from the upgrade and evolution of current systems to the development of new systems.

The initial release of the *Software Technology Reference Guide* is a prototype to provide initial capability, show the feasibility, and examine the usability of such a document. This prototype generally em-

(please turn over)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ■ SAME AS RPT ☐ DTIC USERS ■	21. ABSTRACT SECURITY CLASSIFICATION Unclassified, Unlimited Distribution	
22a. NAME OF RESPONSIBLE INDIVIDUAL Thomas R. Miller, Lt Col, USAF	22b. TELEPHONE NUMBER (include area code) (412) 268-7631	22c. OFFICE SYMBOL ESC/AXS (SEI)

ABSTRACT — continued from page one, block 19

phasizes software technology of importance to the C4I (command, control, communications, computers, and intelligence) domain. This emphasis on C4I neither narrowed nor broadened the scope of the document; it did, however, provide guidance in seeking out requirements and technologies. It served as a reminder that this work is concerned with complex, large-scale, distributed, real-time, software-intensive, embedded systems in which reliability, availability, safety, security, performance, maintainability, and cost are major concerns.