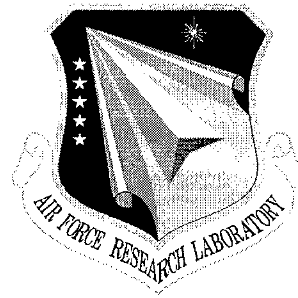
AFRL-IF-RS-TR-1998-194
**Final Technical Report**
**September 1998**




# KNOWLEDGED-BASED SOFTWARE ASSISTANT ADVANCED DEVELOPMENT MODEL (KBSA/ADM)

**Andersen Consulting**

**Chris Faris, Kevin Benner, Junhui Luo, Enaganti B. Naidu, James Fawcett, Benjamin Brunk, Kiran Ganesh, and Udayan Parvate**

*APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED*

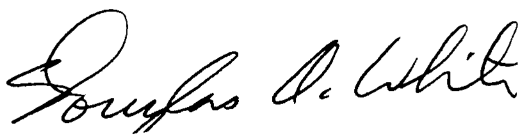
19981210 003

**AIR FORCE RESEARCH LABORATORY**
**INFORMATION DIRECTORATE**
**ROME RESEARCH SITE**
**ROME, NEW YORK**

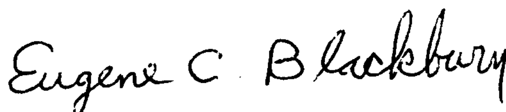
DTIC QUALITY INSPECTED 4

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-1998-194 has been reviewed and is approved for publication.

APPROVED: 

DOUGLAS A. WHITE
Project Engineer

FOR THE DIRECTOR: 

EUGENE C. BLACKBURN
Chief, Information Technology Division
Information Directorate

If your address has changed or if you wish to be removed from the Air Force Research Laboratory Rome Research Site mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/IFTD, 525 Brooks Rd, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

| REPORT DOCUMENTATION PAGE | Form Approved OMB No. 0704-0188 |
|---|---|

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

| 1. AGENCY USE ONLY *(Leave blank)* | 2. REPORT DATE | 3. REPORT TYPE AND DATES COVERED |
|---|---|---|
| | Sep 98 | Final Dec 92 - Jul 97 |

**4. TITLE AND SUBTITLE**

KNOWLEDGE-BASED SOFTWARE ASSISTANT ADVANCED DEVELOPMENT MODEL (KBSA/ADM)

**5. FUNDING NUMBERS**

C - F30602-93-C-0015
PE - 63728F
PR - 2532
TA - 01
WU - 40

**6. AUTHOR(S)**

Chris Faris, Kevin Benner, Junhui Luo, Enaganti B. Naidu, James Fawcett, Benjamin Brunk, Kiran Ganesh and Udayan Parvate

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**

Andersen Consulting
Center for Strategic Technology Research
3773 Willow Rd
Northbrook, IL 60062-6212

**8. PERFORMING ORGANIZATION REPORT NUMBER**

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**

AFRL/IFTD
525 Brooks Rd
Rome, NY 13441-4505

**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

AFRL-IF-RS-TR-1998-194

**11. SUPPLEMENTARY NOTES**

AFRL Project Engineer: Douglas A. White, IFTD, (315) 330-2129

**12a. DISTRIBUTION AVAILABILITY STATEMENT**

Approved for public release; distribution unlimited.

**12b. DISTRIBUTION CODE**

**13. ABSTRACT** *(Maximum 200 words)*

This document constitutes Andersen Consulting's Final Report on the project, the Knowledge-Based Software Assistant/Advanced Development Model (KBSA/ADM). The KBSA/ADM project significantly explored tools and techniques for solving the software development crisis -- improving the productivity, quality, and reliability of software development activities. In particular, the KBSA/ADM explored solutions to these quality of service properties by examining them in the context of: managing the complexity inherent in software development activities; enhancing coordination especially among large development teams composed from different skills sets, stakeholders, experience levels etc., and automating what is understood about the software development process in order to enhance human performance. The KBSA/ADM project conducted by Andersen Consulting has been insightful and fruitful. The results of the project team's efforts begin to implement and describe how KBSA/ADM technologies begin to address the software development problems associated with complexity, automation, and coordination. The most innovative aspects of this work have been in the areas of contextual knowledge (i.e., discussion databases and object linking), process support (i.e., personalized agendas and process enactment), evolution transformations (i.e., transformations which automate stereotypical changes to a model), and critics (i.e., integrating intelligent analysis with process enactment). While significant progress has been made, there are many areas for additional research and expansion on the work presented here.

| 14. SUBJECT TERMS | 15. NUMBER OF PAGES |
|---|---|
| knowledge-based software, software engineering, knowledge-engineering, artificial intelligence, environments, specification languages, requirements analysis, object-oriented | 176 |
| | 16. PRICE CODE |

| 17. SECURITY CLASSIFICATION OF REPORT | 18. SECURITY CLASSIFICATION OF THIS PAGE | 19. SECURITY CLASSIFICATION OF ABSTRACT | 20. LIMITATION OF ABSTRACT |
|---|---|---|---|
| UNCLASSIFIED | UNCLASSIFIED | UNCLASSIFIED | UL |

Standard Form 298 (Rev. 2-89) (EG)
Prescribed by ANSI Std. 239.18
Designed using Perform Pro, WHS/DIOR, Oct 94

## TABLE OF CONTENTS

EXECUTIVE SUMMARY ........ 1

RESEARCH OVERVIEW ........ 3

PROJECT APPROACH ........ 3

KBSA/ADM PROJECT STATEMENT OF WORK ........ 3

KBSA/ADM Architecture ........ 4

KBSA/ADM Process Support ........ 6

KBSA/ADM Requirements Capture ........ 6

KBSA/ADM Modeling and Implementation ........ 7

TECHNICAL ACHIEVEMENTS ........ 9

KBSA/ADM ARCHITECTURE ........ 10

INTEGRATED PROCESS SUPPORT ENVIRONMENT (IPSE) ........ 13

REQUIREMENTS ACQUISITION SUPPORT ENVIRONMENT (RASE) ........ 14

ARGO LANGUAGE ENVIRONMENT (ALE) ........ 14

SESSION AND AGENDA MANAGER (SAM) ........ 15

GRAPH EDITOR FRAMEWORK (GEF) ........ 16

AESTHETIC GRAPH LAYOUT (AGL) ........ 17

SCRIBBLES ........ 17

EXTENDED MODEL VIEW FRAMEWORK (XMVF) ........ 18

REMOTE OPERATIONS (RO) ........ 18

PERSISTENCE AND VERSIONING MECHANISM (PVM) ........ 19

OBJECT LINKING (OL) ........ 19

IMPACT AND INFLUENCE ........ 21

CONCLUSION ........ 24

APPENDIX 1 - Addressing Complexity, Coordination, and Automation in Software Development with the KBSA/ADM ........ 26

Introduction ........ 26

The Problem ........ 26

## TABLE OF CONTENTS (Continued)

A Functional Description of the ADM ........ 27

The Solution ........ 28

Conclusion ........ 36

APPENDIX 2 - RASE: an Integrated Requirements Acquisition Support Environment ........ 38

INTRODUCTION ........ 38

PROBLEM STATEMENT ........ 39

RASE CORE FUNCTIONALITY ........ 40

EXTENDED RASE CAPABILITIES ........ 43

IMPLEMENTATION INFORMATION ........ 45

RELATED WORK ........ 46

CONCLUSION AND FUTURE DIRECTIONS ........ 47

REFERENCES ........ 48

APPENDIX 3 - Evaluation of a Knowledge Based Software Assistant Advanced Development Model ........ 50

TABLE OF CONTENTS FOR APPENDIX 3 ........ 52

## TABLE OF FIGURES

Figure 1 - Prototypical Software Development Life Cycle with KSBA/ADM Vision Overlay ................ 2

Figure 2 - KSBA/ADM Capabilities Relative to Software Development Problem Areas ........................ 4

Figure 3 - KSBA/ADM Session-View Paradigm Architecture ................................................................11

Figure 4 - KSBA/ADM Intra-Tool Architecture ..................................................................................... 12

Figure 5 - Process to Process Communications Flows ........................................................................... 16

Figure 6 - ADM Common Services, Tools, and Development Activities .............................................. 28

Figure 7 - ADM Session and Agenda Manager ..................................................................................... 32

Figure 8 - The hyper document editor / browser tool ............................................................................ 41

Figure 9 - The REMAP Model ............................................................................................................... 42

Figure 10 - The REMAP tool user interface .......................................................................................... 42

Figure 11 - The Term Dictionary tool user interface ............................................................................. 43

Figure 12 - KSBA - Architecture .............. .......................................................................................... 50

Figure 13: **Project Knowledge Structure** ........................................................ 104

Figure 14: **Conceptual Model** ............................................. 105

## Executive Summary

This document constitutes Andersen Consulting's Final Report on its project, the Knowledge-Based Software Assistant/Advanced Development Model (KBSA/ADM). Instantiated under contract F30602-93-C-0015 by Rome Laboratory in 1992, the KBSA/ADM project significantly explored tools and techniques for solving the software development crisis ---- improving the **productivity**, **quality**, and **reliability** of software development activities. In particular, the KBSA/ADM explored solutions to these specific "ilities" by examining them in the context of:

- Managing the **complexity** inherent in software development activities,

- Enhancing **coordination** especially among large development teams composed from different skills sets, stakeholders, experience levels etc., and

- **Automating** what is understood about the software development process in order to enhance human performance.

Focusing on these areas required a holistic, full software development life cycle approach and addressed several key areas:

1. **Methodology and Project Process Definition** - The ability to define, manage, and repeat a comprehensive software development process is recognized industry wide as a key to successful software development. The KBSA/ADM uses methodology and project process definition, instantiation, and enactment as a back bone to all other KBSA/ADM tool processes, yet still allows for flexibility in responding to the dynamic nature of an on-going project to new information, tasks, changing personnel, etc.

2. **Requirements Capture** - Simply building a system without adequately capturing the requirements, decisions, and discussions predicating the system is a recipe for disaster. Requirements capture within the context of a project process was a focal point of the KBSA/ADM project. Requirements and supporting documentation in the commercial software development arena takes on many forms: meetings, informal discussions, memos, drawings, recordings etc. In the KBSA/ADM this knowledge is referred to as *formal* and *informal* knowledge. The KBSA/ADM project focused extensively on the capture and management and tracability of this knowledge throughout the life cycle through the use of tools and *evolutionary transformations* (ETs) .

3. **System Modeling** - Once requirements are captured, the definition of a software system transforms or evolves through a series of modeling mutations until the final system is deployed (and in fact, in the maintenance phase of the software life cycle, continues to evolve). Continuous refinement of these models, linking these refinements back to original requirements and early concepts was the goal of the KBSA/ADM. As a system evolves to a state of executing computer code, these refinements are continuously captured and evolved.

Graphically, the complexity and dynamic nature of the software development process is represented in figure 1.

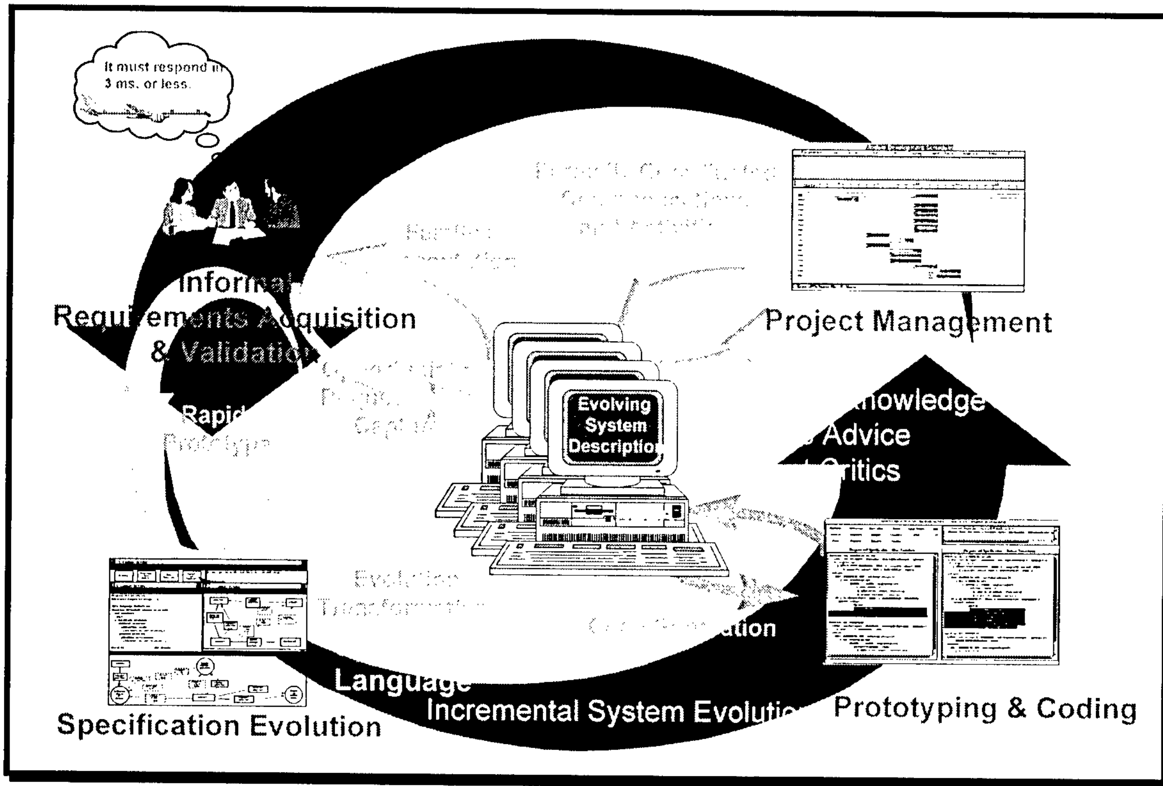



*Figure 1 - Prototypical Software Development Life Cycle with KBSA/ADM Vision Overlay.*

It is within the above three areas that the KBSA/ADM researchers and development team focused its efforts and research activities. The KBSA/ADM team extended work first articulated in a vision for the KBSA program in "Report on a Knowledge-Based Software Assistant"[Gr83] in 1983. Since then, Rome Laboratory has funded a variety of research efforts which have made significant steps toward realizing and refining this vision ([Sm93], [Jo93], and [De92] among several others). Today, Andersen Consulting's Center for Strategic Technology Research (CSTaR) field-able prototype of a KBSA called the Knowledge-Based Software Assistant / Advanced Development Model (or ADM for short) embodies many of the results of preceding research efforts, as well as refines the KBSA vision as it relates to addressing today's software development problems.

It is not enough, however, to focus primarily on these research areas without considering the unifying "glue" which holds them together. At the functional level, software development process and methodology provide the guidance by which team members perform their assigned roles. At the technical level, a unifying architecture ensures the suite of KBSA/ADM capabilities forms a consistent, extensible *federation* of cooperating tools to deliver value on the KBSA/ADM vision.

# Research Overview

## *Project Approach*

Andersen Consulting's formal involvement on the KBSA/ADM consisted of two phases:

**Phase 1 - Initial Exploration**: This effort focused on the examination and exploration of emerging technologies suitable for incorporation into a KBSA/ADM integration effort. Specifically, they included: software selection (e.g. Objectstore OODBMS, Galaxy Graphical User Interface library, Sun Solaris, C++ etc.); infrastructure building, including the development of the Aesthetic Graph Layout (AGL) and Scribbles libraries; research examination (IBIS/REMAP understanding, as well as Module Specification Language – i.e. ARGO); and initial prototyping (e.g. early releases of project management tool, REMAP, and Module Specification Language Environment).

**Phase 2 - Field-able Prototype Roll-out:** This effort expanded on "lessons learned" from Phase 1 to incorporate existing functionality and new KBSA/ADM tool capability onto a common and extensible technical architecture. During Phase 2 the project team began collaborations with additional software engineering research efforts across Andersen Consulting including the "Component-based Software Engineering (CBSE)" project, Foundation, and AC Design. The KBSA/ADM's influence on these efforts is outlined in the **Impact and Influence** section of this document.

## *KBSA/ADM Project Statement of Work*

An examination of the KBSA/ADM Statement of Work (SOW) identifies some overriding areas of research focus. With respect to the KBSA/ADM research effort, the project's researchers and developers have focused on some significant areas of the software development life cycle relative to key SOW requirements. In this section, a correlation of these research areas, their significant contributions and results is made with the SOW.

The KBSA/ADM project team, during Phase 2 of the project, identified figure 2 as a high-level model to approach the software development problems articulated in the **Executive Summary** section

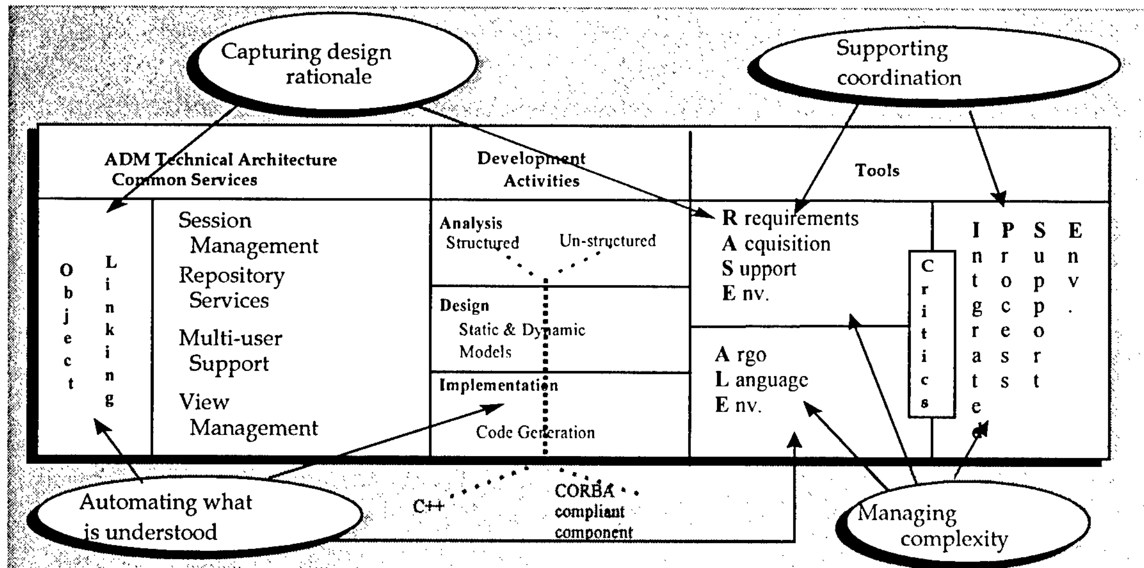



*Figure 2 - KBSA/ADM Capabilities Relative to Software Development Problem Areas.*

From figure 2, across the a typical software development life cycle (e.g. Analysis, Design, Implementation), an architecture and suite (i.e. the "federation") of KBSA/ADM tools was identified to attack the issues of complexity, coordination, and automation. These problem areas are also identified in the KBSA/ADM Statement of Work (SOW) as particular areas of emphasis.

## KBSA/ADM Architecture

The KBSA/ADM SOW is particularly emphatic with respect to the following:

- "...the KBSA shall consist of an integrated and uniform design...[SOW -p. 3]"
- "The KBSA shall be designed to be flexible and extensible...[SOW p. 3)]"
- "Primary concerns are the completeness, quality, and maturity of technology and products to be used or produced and the overall usability of the ADM...[SOW p. 7]

KBSA/ADM project researchers and developers spent considerable effort in ensuring the achievement of these three requirements. As of delivery of the KBSA/ADM prototype, these three objectives have been met:

1. The suite of KBSA/ADM capabilities is constructed on a unifying technical architecture. Composed of a Session-View paradigm built around a central, object-oriented database design repository, the KBSA/ADM architecture allows for tools to be developed independently of each other, yet achieve a high degree of collaboration required to support teams of developers across the software development life cycle. Built from a set of object-oriented frameworks, the KBSA/ADM federation of tools adheres to a unifying method of inter- and intra-tool communication. The **KBSA/ADM Technical Design documentation** outlines the design philosophies, framework descriptions, and current KBSA/ADM tool descriptions from a technical perspective.

2. The software development life cycle is a complex, ever evolving process by which practitioners of software engineering perform their work. It is highly desirable that a KBSA/ADM have a flexible architecture to incorporate unanticipated needs. In fact, as a research project, the KBSA/ADM has evolved over the life of the contract incorporating fresh ideas from project participants. Within this context of flexibility, new ideas, tools, and KBSA/ADM capabilities can and have been incorporated. The Naval Post-Graduate has extended KBSA/ADM requirements capture capability with Truth Maintenance System Extensions (TMS), and Syracuse University has added a completely new tool for the management of source code and test data information. As the evaluator of KBSA/ADM technology, Syracuse extended the KBSA/ADM federation with completely new functionality as opposed to NPS's functional extensions.

3. The KBSA/ADM prototype is built upon state of the market, market leading technologies:
    - **Objectstore** - This object-oriented database management system acts as the underlying DBMS for the design repository. State of the art, Objectstore supports a "virtual plain" of objects, versioning all within a client/server architecture.

    - **Galaxy Graphical User Interface** - This GUI builder supports the development of cross-platform applications using state of the market window-based interfaces. Additionally, Andersen's Aesthetic Graph Layout (AGL) library supports the implementation of sophisticated graphical interfaces.

    - C++ - The leading object-oriented development environment, the KBSA/ADM team developed the prototype using Sun Microsystem's current version of Teamware.

    - **Sun Solaris v2.5 and Windows NT 3.5.1** - These operating systems are technical leaders in their respective markets.

    - **Pure Software Testing Products** - Designed to ease the development of complex software, these tools were invaluable for debugging KBSA/ADM functionality.

    - **Expersoft's Powerbroker Object Request Broker** - Using Powerbroker facilitated the development of communication infrastructure for inter-process communication.

    - **Windows NT Port** - While built upon market leading system software, the attribute inexpensive does characterize to these systems and platforms. In an effort to reduce costs, a prototype port of certain KBSA/ADM functionality demonstrated the feasibility of moving the KBSA/ADM to the Windows NT platform, providing further evidence of the extensibility of the prototype.

Clearly the decisions made by the KBSA/ADM development team and researchers have positioned the KBSA/ADM for the future. In terms of innovation, the KBSA/ADM technical architecture provides a platform for additional research, a platform for increasing KBSA/ADM robustness, extending into new areas of research, and a platform for enhancing existing ideas already implemented.

## KBSA/ADM Process Support

To support the collaboration of a software development team, both among its members and with outside customers, stakeholders, and sponsors, a development process must be defined to assure success. The KBSA/ADM SOW clearly defines the objectives regarding the requirement for process support:

- "Formal coordination of activities and communications enabling automation of the software life cycle processes [SOW p. 8]"
- "Project management for all management levels including assistance for project planning, analysis... [SOW p. 8]"
- "Support for application developments of 50KLOC [SOW p. 9]"
- "Support for simultaneous and interactive use by a development team consisting of 1 to 4 developers. [p. 9]".

With respect to these requirements, the KBSA/ADM prototype enables a small team of developers to establish a methodology, instantiate project plans from that methodology and be guided by the methodology once enacted. The KBSA/ADM prototype interfaces with Microsoft Project to utilize Project's capabilities for defining projects, tasks, and deliverables. Extensive on-line help assistance at both the methodology and project plan definition level is provided on the Windows NT and UNIX platforms. Critics "hover" over an imported project plan and determine when a task can be enacted by a project team member depending on the state of dependent tasks and work-in-progress.

Limitations in the existing KBSA/ADM prototype restrict the sophistication and complexity of the defined project, but support for 1 to 4 developers creating 50,000 lines of code is achievable. Syracuse University's work in project management is a promising extension to the existing KBSA/ADM project capabilities and should be encouraged. Additionally, CoGenTex's (**www.cogentex.com**) work with the Project Reporter is another interesting extension of natural language technologies to project status reporting.

## KBSA/ADM Requirements Capture

The Statement of Work required the KBSA/ADM make significant advances in the area of requirements capture. The implementation of the KBSA/ADM Requirements Acquisition Support Environment (RASE) functionality meets the following requirements:

- "Acquisition, analysis, integration,...of application system requirements...representations that are appropriate for the application user community. [SOW p. 7]"
- "Acquisition, recording, and management of domain knowledge... reuse and replay. [SOW p. 8]"
- "Capture, maintenance, and generation of documentation in both interactive help and report form. [SOW p. 8]"

RASE supports a sophisticated capability for collection of knowledge in different media and forms. Unstructured discussions of requirements can be ordered using REMAP technology. Information in multi-media formats can be linked into hyper documents capturing textual information. The ability to link information, print information, and relate information to

"upstream" and "downstream" development artifacts of the software life cycle is a RASE strength.

A key "assistance" technology demonstrated in RASE is the implementation of *evolution transformations* (ETs). ETs demonstrate the capability to transform one form of knowledge (e.g. a text phrase) into another semantically related form of knowledge (e.g. an object class) and preserve the contextual link between the concepts. In this way, a developer can begin to capture at a high level, the requirements and concepts underlying the software system and relate them back to earlier concepts. Over time, a lattice of objects and links develops in the design repository that is traceable and analyzable by critics, "visitors" (as proposed by Syracuse University), or humans responsible for further system development.

## KBSA/ADM Modeling and Implementation

The process of transforming requirements into system capabilities is typically an evolution of models to ever greater levels of precision. Executable source code on a computer is the ultimate refinement of a requirement through to an executing, computing process. Determining whether that executable process is correct, adequately meets the requirements, or is error free is dependent on the quality of the requirement and the transformation processes placed on the intermediate models.

The KBSA/ADM has focused on modeling, the transformation of information, and the linking of related concepts in order to illustrate capabilities at the prototype level. Specifically in the KBSA/ADM SOW, modeling and model transformation capabilities are describes as follows:

- "Incremental elaboration, evolution, analysis, refinement and validation of formal specification... [SOW p. 8]"
- "Optimization, generation, and targeting of application software from formal specifications. [SOW p. 8]
- "Ability to produce software systems written in C/C++ [SOW p. 9]"
- "Assistance and Generation of test data and programs. [SOW p. 9]"

The KBSA/ADM, through its ARGO Language Environment (ALE) component demonstrates a capability for these areas by modeling concepts introduced using evolution transformations from RASE (concept-to-class transformation) in an OMT-like notation. ALE provides rudimentary language and modeling capabilities with the ability to generate C++ header files and link to requirements documented in RASE. The underlying ARGO language semantic checker performs rudimentary semantic checking of a model, however additional research remains in this area. Complementing ALE's ability to model software system concepts is CoGenTex's Model Explainer technology (refer to **www.cogentex.com**). Using a Web-based HTML generator, Model Explainer "reads" ARGO specifications to generate a natural language interpretation (with examples) of the model. Ideal for review and discussion with stakeholders and other non-technical personnel (e.g. review participants not versed in ARGO language or OMT-like object-oriented modeling notation), the Model Explainer provides an alternative "view" onto a model, providing added insight.

Syracuse University has extended the KBSA/ADM suite of tools with a Project Archival and Report Tool (PART) to more strongly augment the relationship between software program deliverables (e.g. code modules, test data) and the project plan tasks creating these deliverables. PART augments some of the frailties in this implementation of ALE by allowing source code to be imported directly into the KBSA/ADM repository and treated as any other design artifact. Using object linking and hyper document repository topics, source code can be annotated with additional semantic information and models.

The particular KBSA/ADM Statement of Work accomplishments are elaborated in the next section, **Technical Achievements**.

## Technical Achievements

The technical goals of the KBSA/ADM project are to research and develop the various technical deliverables. The key technical deliverables include:

- **KBSA/ADM Architecture:** an architectural framework that supports requirements of various types of stakeholders such as the End Users, Tool Developers, System Architects and Project Management involved in the software development process. It is an open, distributive, collaborative and customizable architecture, and developed based on the principle of federation of cooperating processes.

- **Integrated Process Support Environment (IPSE) :** a KSBA/ADM facility that supports the process and methodology functionality including project plan definition, task assignment, task enactment, and collaboration.

- **Requirements Acquisition Support Environment (RASE):** a KBSA/ADM facility that addresses the "upstream" life-cycle issues such as capturing informal systems descriptions and design rationale in a software development process.

- **ARGO Language Environment (ALE):** a KBSA/ADM facility that supports the "downstream" life-cycle stages such as design and maintenance in software development. ARGO is a KBSA/ADM language for the specification and implementation of object-oriented systems.

- **Session and Agenda Manager (SAM):** forms the hub of a KBSA/ADM environment, and manages the inter tool cooperation and provides the session-view paradigm.

- **Graph Editor Framework (GEF):** provides a virtual and customizable diagrammatic interface through automatic visualization and manipulation processes.

- **Aesthetic Graph Layout (AGL):** provides automatic graph layout capabilities including node positioning, moving and re-sizing, automatic edge graphs and multiple layout routing.

- **SCRIBBLES:** provides the capabilities to visualize and interact with graphical objects such as rectangles and lines.

- **eXtended Model View Framework (XMVF):** provides intra-tool message based communication between the front-end (views) and the back-end (model) of a tool.

- **Remote Operations (RO):** provides inter-tool message based communication across a network of KBSA/ADM facilities.

- **Persistence and Versioning Management (PVM):** defines the KBSA/ADM repository and provide version management (check in/out) and project management utilities.

- **Object Linking (OL):** implements the relationship that cross KBSA/ADM work objects and process boundaries.

The rest of section describes all the technical deliverables in detail.

## *KBSA/ADM Architecture*

**Key technical elements:** KBSA/ADM environment provides an architectural framework that supports requirements of various types of stakeholders such as the End Users, Tool Developers, System Architects and Project Managers involved in the software development process. It is developed based on the principle of federation of tools rather a monolithic system. It is an open, distributive, collaborative and customizable architecture.

The KBSA/ADM architecture is a three-tiered model consisting of the session-view layer, tool layer and repository layer as shown in figure 3. The session-view layer represents the end user perception of KBSA. The tool layer represents the inter connection (a star network of KBSA facilities with SAM as a hub) among KBSA/ADM facilities such as ALE, RASE, IPSE etc. The repository layer consists of back end facilities such as the Objectstore server for managing persistent objects.

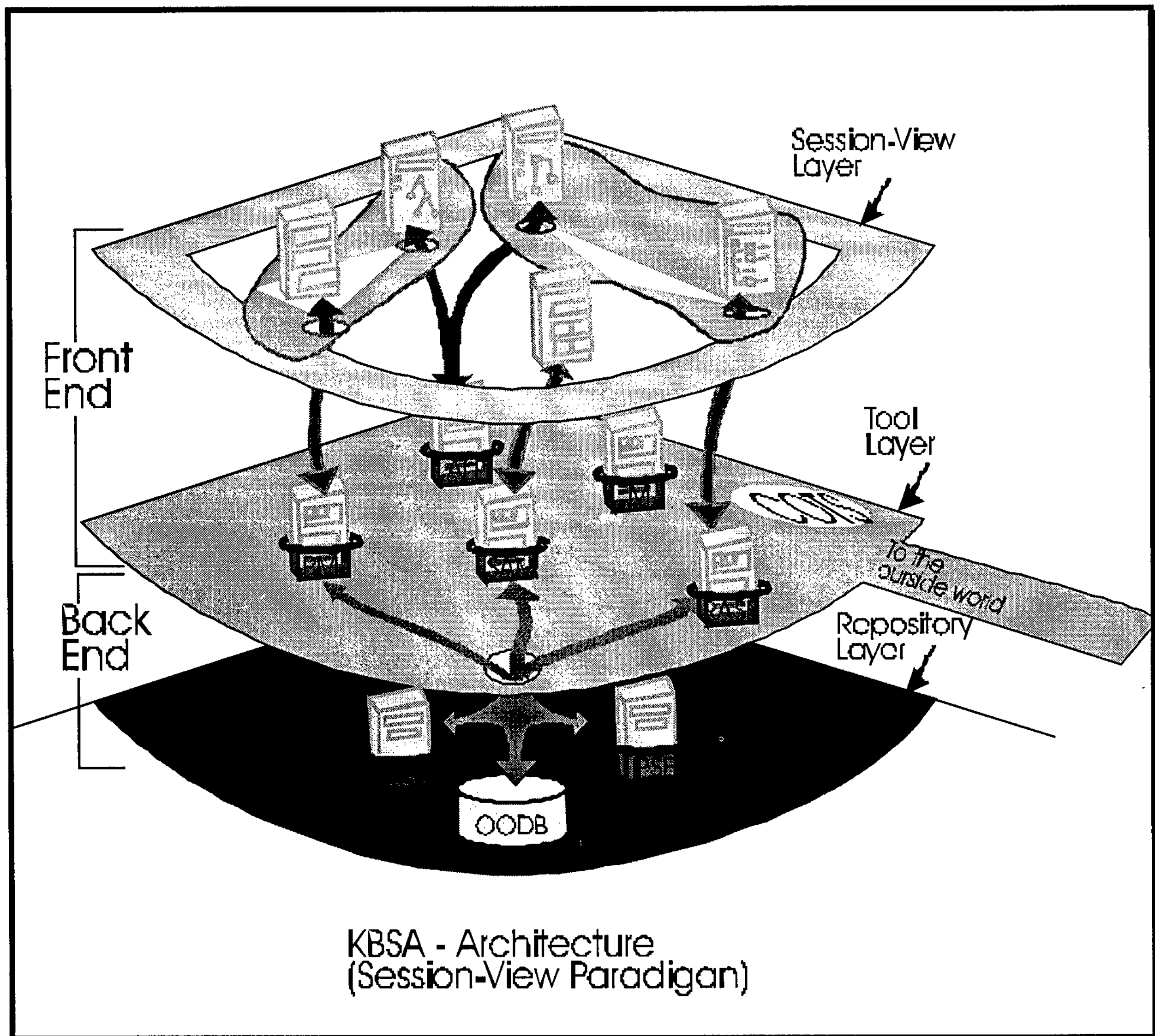



*Figure 3 - KBSA/ADM Session-View Paradigm Architecture*

The KBSA/ADM architecture provides session-view paradigm, message based communication, and framework based tool development. A Session-View paradigm provides the end users (developers) with a perception of the KBSA/ADM as a seamless development environment rather than a set of distinctly independent tools. Message based communication is provided to de-couple the tools from each other, and views from the model in a tool.

The KBSA/ADM architecture proposes framework based tool development for effective reuse by reducing a tools research and development effort. The KBSA/ADM tool architecture consists of three layers of frameworks including Presentation Layer, Messaging Layer, and Repository Layer, as shown in figure 4. The Presentation Layer frameworks are responsible for empowering the end user to perform the automatic visualization and manipulation of work objects. The Messaging Layer frameworks are responsible for providing inter- and intra-tool communication.

The Repository Layer frameworks are concerned with the persistence and version management of work objects in the KBSA/ADM repository, and implementing the relationship between persistent objects.

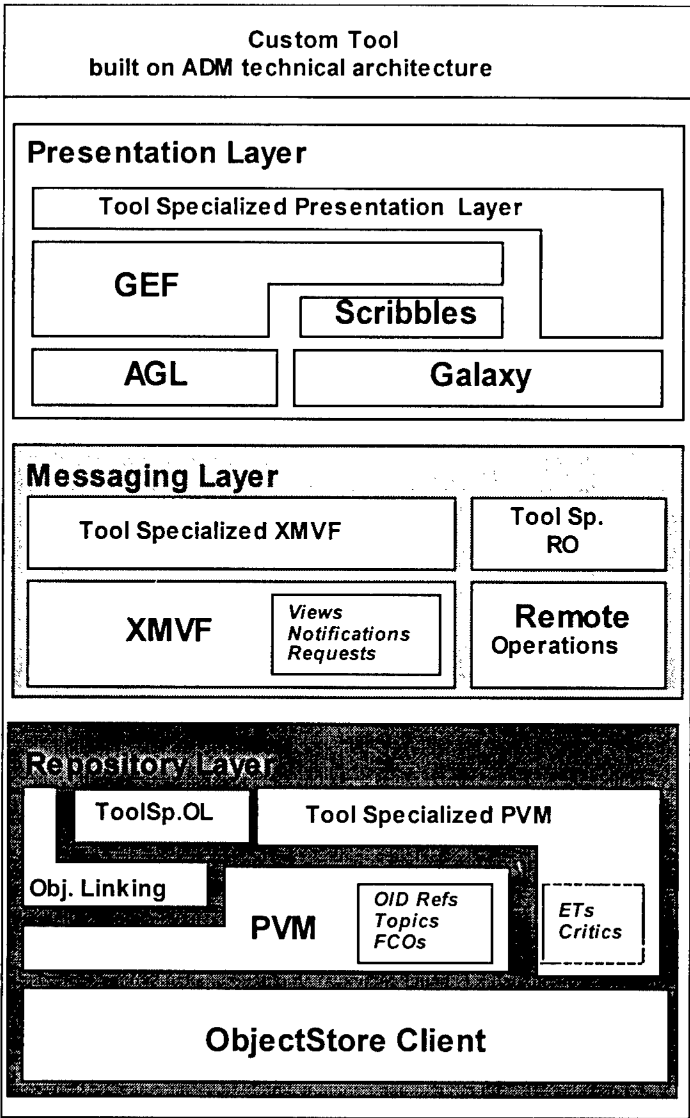



*Figure 4 - KBSA/ADM Intra-Tool Architecture*

**Innovative ideas:** The Session-View paradigm is a fundamental design axiom that enables an intuitive interface for software developers to understand their requirements in terms of the elements of a software development methodology (such as requirements, class diagrams etc.). Hence the evolutionary changes to the KBSA/ADM environment does not affect the user's (developers) perspective of the environment in the session-view paradigm. The three tier architecture enables the end users (i.e. "developers") to extend and customize the environment to their requirements. Architecture openness has been demonstrated successfully through the KBSA/ADM COTS interface to Microsoft Project and PART (Project Archival and Report Tool)

as part of the KBSA/ADM evaluation. The KBSA/ADM provides a seamless protocol for inter-tool communication.

Framework based tool development adds substantial value to the process of KBSA/ADM tool development. The main advantage of the frameworks include software reusability (higher productivity, better portability, and higher reliability) and ease of enforcement of a uniform interface standard. The challenge of developing reusable frameworks is effectively demonstrated during the development of the KBSA/ADM architecture.

**Status:** The KBSA/ADM architecture is fully developed and tested. It provides multi-user capability effectively. In the current implementation, the IPSE is linked to the Session and Agenda Manager (SAM) as one process.

**Areas for improvement:** The KBSA/ADM login functionality is primitive - not allowing re-login as a different user or login into another database from the current login. The session-view management can be further improved with a graphical interface to projects, tasks, users, deliverables and the relationship among them.

## *Integrated Process Support Environment (IPSE)*

**Key technical elements:** The Integrated Process Support Environment (IPSE) is a KBSA/ADM facility (tool) that addresses the process and methodology related issues in software development. IPSE supports a methodology independent and process driven approach facilitating the management of the software development process. It provides collaboration in KBSA through automated task definition, tracking and resolution support. IPSE's meta model includes the concepts of project plan, task, resource, deliverable, dependency and resolution. IPSE supports isomorphic representation of project plans through a tight integration with MS Project. The integration between IPSE and MS Project is provided through the KBSA/ADM commercial off-the-shelf (COTS) tool interface.

**Innovative ideas:** The IPSE design supports a neutral representation (KBSA independent) of project plans through the COTS interface with MS Project. The KBSA/ADM's usage of methodology independence, as demonstrated by integrating Andersen Consulting's Object Design Methodology (ODM) deliverables, demonstrates this methodology independent capability . One of the challenges in IPSE is ensuring the project consistency both on-line and off-line across users. Some hard issues like automatic task enactment, automatic generation of users tasks, automatic propagation of critic resolution tasks have been successfully addressed.

**Status:** The IPSE tool is fully functional. Usability analysis and initial testing have been performed successfully. The current version IPSE supports a limited functionality in resource allocation and time management.

**Areas for improvement:** Advanced Critic resolution task management, graphical representation of project plan, and better project merge/synchronization features are some of the areas for future research.

## *Requirements Acquisition Support Environment (RASE)*

**Key technical elements:** The Requirements Acquisition Support Environment (RASE) is a KBSA/ADM facility (tool) that addresses the "upstream" life-cycle issues in software development. RASE supports the capturing of an informal system description in natural language, such as user interview transcripts, memos, requirements or design documents in the form of hypertext documents; and semi-formal descriptions such as design rationale (i.e. Why it is the way it is?) in the form of structured design discussions.

**Innovative ideas:** RASE provides a hypertext document editor/browser for capturing systems descriptions in natural language. It provides an enhanced IBIS/REMAP structured discussion model to capture design rationale. REMAP is further enhanced with Truth Maintenance capabilities, including the automatic validation of dependencies between discussion nodes. The user is alerted if previous decisions must be re-evaluated through automatic propagation of decision dependencies. Across all of RASE views, object linking capability is enabled to link design information to various life cycle artifacts in the KBSA/ADM environment. Additionally, advanced evolution transformations (ETs) are provided to automate mundane analysis and development tasks across tools.

**Status:** The RASE tool is fully functional including the truth maintenance capability. It provides a general purpose and reusable Hypertext library for other tools to incorporate hypertext capability if needed. Usability analysis and detailed testing have been performed successfully.

**Areas for improvement:** The future research areas in RASE include extending the structured discussion component with World-Wide-Web based user interface and the concept of stakeholder.

## *ARGO Language Environment (ALE)*

**Key technical elements:** The ARGO Language Environment (ALE) is a KBSA/ADM facility (tool) that supports the "downstream" life-cycle stages such as design and maintenance in software development. It provides an environment for the specification and evolution of object-oriented specifications in the ARGO specification language. ARGO is the KBSA/ADM language for the specification and implementation of object-oriented systems. It supports the concept of attributes and relations to specify the object structure, and class invariant and pre- and post-conditions on member functions. ALE's meta model is represented using the ARGO language. ALE provides critics that evaluate the specification and propose changes that could improve its quality. It also provides c++ code generation capability for the design components.

**Innovative ideas:** ARGO supports Meyer's notation of "Design by Contract" explicitly. It attempts an enhanced version of C++ with a simplified object declaration syntax. ARGO is influenced by ODMG's ODL and OMG's IDL. It is implemented using PCCTS (Purdue

Compiler Construction Tool Set) - a public domain software tool similar to YACC. ALE is empowered with the ideas of critics and evolution transformations. Critics are the best manifestation of the assistant metaphor in the KBSA. The work on critics draws from that of Fischer, whose goal was to automate mundane tasks and help bring to bear knowledge at the appropriate time and location when the users needed it. Evolution Transformations are editing operations which make complete semantic changes to a model.

**Status:** ALE is presently the weakest tool in the KBSA/ADM federation of tools. Usability analysis and initial testing have been performed identifying significant areas for enhancement.

**Areas for improvement:** Future research areas in ALE include enhancing the ARGO language and its front-end graphical support, developing a general purpose framework for critics, advanced evolution transformations, and tighter integration with CoGenTex's Model Explainer technology.

## *Session and Agenda Manager (SAM)*

**Key technical elements**: The KBSA/ADM facilities are constructed as a federation of cooperating processes. The Session and Agenda Manager (SAM) manages the inter tool cooperation and provides the session-view paradigm. SAM is one of the KBSA/ADM facilities built on the same tool technical architecture as are the other KBSA/ADM facilities such as ALE, RASE etc. The underlying design principles of SAM are to cooperate with the facilities in providing inter-tool communication and to enable smooth integration and customization of facilities (old or new) with the KBSA/ADM environment. SAM forms the hub of the KBSA/ADM facilities network as shown in figure 5. Upon login, SAM automatically connects to other instances of SAMs (KBSA/ADM instances) that are logged into its repository.

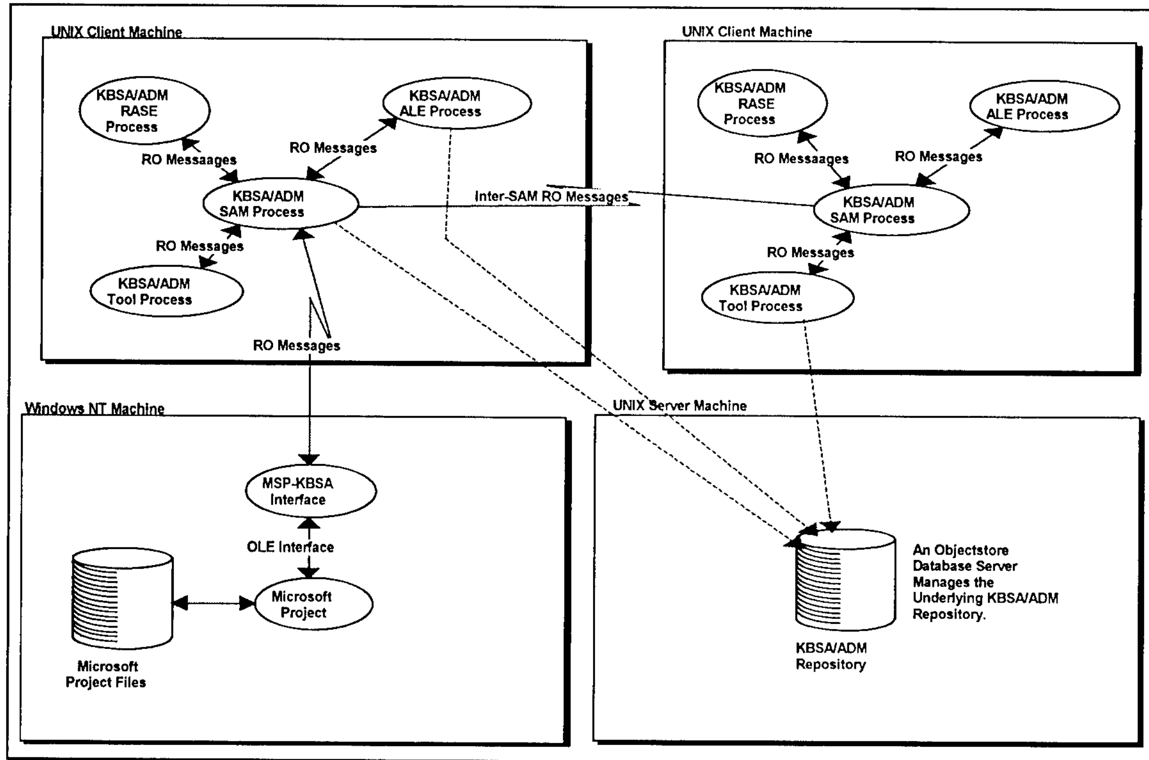



*Figure 5 - Process to Process Communication Flows*

**Innovative ideas**: The session-view paradigm successfully demonstrates the required level of transparency to the end-user working in the KBSA/ADM environment. Sessions, saving session context on exit, and restoration of session context upon re-login are some of the ideas that add a great value to the end-user. The concept of on-line registration of KBSA/ADM facilities paved the way to the smooth integration and customization of the environment. The concept of a transportation layer in the remote messages enabled the effective de-coupling of the facilities. Automatic connection to other related SAMs enabled effective collaboration among a team of users.

**Status**: SAM is fully functional. Usability analysis and detailed testing have been performed successfully.

**Areas for improvement**: A graphical user interface to a session and its contents would greatly improve usability. Usability would be greatly improved with the ability to login as a different user or to another repository without the need to log out of and restart the KBSA/ADM.

## Graph Editor Framework (GEF)

**Key technical elements**: The Graph Editor Framework (GEF) is one of the frameworks of the Presentation layer. It provides a virtual and customizable diagrammatic interface through visualization and manipulation processes. It encapsulates AGL and SCRIBBLES frameworks and makes them as transparent as possible for developers thereby accelerating diagrammatic interface

development. The GEF is developed over Galaxy - a platform independent GUI development environment.

**Innovative ideas**: The challenge of developing a reusable framework is proved effectively through this framework. GEF set the standards in defining the framework interfaces and addressed the issues relating to integration of frameworks while developing a tool. One of the key ideas in GEF is to identify the two independent processes in graph editing namely visualization and manipulation. This framework reduced the development effort considerably (10-to-1 code reduction and reduced the development time by one-third).

**Status**: This framework is fully developed, thoroughly tested and effectively reusable. Currently it is reused across three projects - KBSA/ADM, Component-based Software Engineering and the Naval Post-Graduate School-KBSA . The required reference and user manuals are available. It is available both on Unix and NT.

**Areas for improvement**: GEF currently depends on Galaxy. However, supporting it with the same unified interface on other popular platforms such as Microsoft Foundation Classes(MFC) is a great challenge. GEF currently has a limited support for enforcing restrictions between graphical objects. This is one research area where a simplified and code free interface is required to enforce restrictions between objects.

## *Aesthetic Graph Layout (AGL)*

**Key technical elements**: Aesthetic Graph Layout (AGL) is one of the frameworks of Presentation Layer. It provides automatic graph layout capabilities including node positioning, moving and resizing, automatic edge routing, nested graphs and multiple layouts: tree, hierarchy, network and manual.

**Innovative ideas**: As part of this framework, a number of new algorithms are developed. These include automatic edge routing, node placing, space creation and compaction, and static graph layout for network layout. Also developed was a test framework to visualize the graph layout process. Providing a layout neutral interface is an idea that enhanced usability the AGL considerably.

**Status**: It is fully developed, thoroughly tested and reusable effectively. It is currently used by GEF. It does not depend on any commercial software.

**Areas for improvement**: Extending the AGL concepts to other diagrammatic domains such as VLSI layout and CAD/CAM applications is a new research area. Enhancing the AGL with manual operations such as updating the edge orientation is challenging task for future research.

## *SCRIBBLES*

**Key technical elements**: SCRIBBLES is one of the frameworks of the Presentation layer. It provides the capabilities to visualize and interact with graphical objects such as rectangles and lines. It is an extension of Galaxy's graphical features.

**Innovative ideas**: Galaxy provides very primitive facilities for graphical objects. Scribbles effectively encapsulates the Galaxy graphical facilities and defines various objects and operations on them effectively.

**Status**: It is fully developed, thoroughly tested and reusable effectively. It is available for both Unix and Windows NT platforms.

**Areas for improvement**: SCRIBBLES fully depends on Galaxy. However, supporting it with the same unified interface on other popular platforms such as MFC is a great challenge.

## *eXtended Model View Framework (XMVF)*

**Key technical elements**: The eXtended Model View Framework (XMVF) is one of the frameworks of the Messaging layer. It is a modified model-view-controller framework often found in Smalltalk environments. The key objectives of this framework include establishing and maintaining the relationships between the front-end (views) and back-end (model) of a tool, and providing a consistent mechanism for defining messages between the views and the model of a tool. The communication between the views and model is provided through messages called requests and notifications.

**Innovative ideas**: De-coupling of views from the model is effectively implemented. It encapsulates the transaction management functionality and distributions of notifications to the related views.

**Status**: It is fully developed and thoroughly tested. It is difficult to use XMVF without Object Store and PVM functionality.

**Areas for improvement**: Current implementation of XMVF depends on the back-end (Object Store) functionality. This is one of the bottlenecks in reusing the framework effectively. An improved filtering mechanism over the notification would add value to XMVF functionality. Providing publish and subscribe mechanisms between views and models in some cases is also desirable.

## *Remote Operations (RO)*

**Key technical elements**: The Remote Operations (RO) is one of the frameworks of the Messaging layer. It is built upon Expersoft's Powerbroker object request broker framework. RO provides inter-tool communication across a network of KBSA/ADM facilities. The communication between KBSA/ADM facilities is provided through messages (knows as Abstract Data Types -ADTs). RO provides communication between platforms such as UNIX and Windows NT.

**Innovative ideas**: This framework completely encapsulates Powerbroker thereby providing flexibility to migrate to other industry standard object brokers if needed. RO supports synchronous communication between processes. One of the strengths of this framework is its simplicity in the interface (two functions – send and process a message).

**Status**: This framework is fully developed and thoroughly tested. It is reusable with little effort.

**Areas for improvement**: Enhancing the RO with Publish and Subscribe mechanism across processes would be an interesting research area. RO currently uses ports for inter process communication. This mechanism is very primitive and not CORBA compliant. Migrating to an advanced communication mechanism is desirable to reduce complexity. Support for nested messages is another area to be addressed.

## *Persistence and Versioning Mechanism (PVM)*

**Key technical elements**: The Persistent and Versioning Management (PVM) is one of the frameworks of the Repository Layer framework. PVM defines the KBSA/ADM repository and provides version management (check in/out) and project management utilities. Built over an object-oriented database management system called ObjectStore, PVM encapsulates the required functionality of ObjectStore for basic persistent and versioning. It defines various XMVF requests and notifications to update the persistent objects.

**Innovative ideas**: One of the challenges in developing PVM is to define the KBSA/ADM repository. We overcome the challenge by defining the two-tiered workspaces - project and session, basic versioning particle (topic) and basic persistent particle (first class object).

**Status**: It is fully developed and tested. It is reusable with some additional effort.

**Areas for improvement**: Support for sub-project and project base-lining are two areas that can add greater value to the KBSA/ADM. Adding a repository manager that browses the database and allows some simple editing features over the persistent objects is desirable for effective repository management. Support for alternative design options is another future research area.

## *Object Linking (OL)*

**Key technical elements**: Object Linking is also a framework of the Repository layer. It implements relationships that cross KBSA/ADM work objects and process boundaries. It supports the ability to create, traverse and delete cross-work object relationships. It uses Remote Operations (RO) functionality internally. The object linking is the key mechanism for providing the ability to link design artifacts and understand the relationship between artifacts as and when needed.

**Innovative ideas**: Encapsulating the object link capability within the first class object is a great idea that provided flexibility in linking design artifacts of different kinds. Distributing object links into two halves between their respective objects proved to be a feasible solution.

**Status**: It is fully developed and partially tested. It is reusable with little effort.

**Areas for improvement**: Enhancing of object links with semantics would provide opportunity for future research. Handling broken links with contextual information would add great value to the end user.

## Impact and Influence

In Syracuse University's "Evaluation of a Knowledge Based Software Assistant Advanced Development Model", they state: "There is a lot to like about the KBSA/ADM." With respect to this statement, Andersen's KBSA/ADM research and development team spent considerable resources to engage and support those interested in the technical areas the research was exploring. Specifically, the KBSA/ADM team focused on the following organizations to facilitate technology transfer:

- Andersen Consulting
  - FOUNDATION
  - AC Design
  - Architecture and Tools Program
  - Component-Based Software Engineering (CBSE)
  - Technology Reinvestment Program (TRP)
- CoGenTex, Inc.
- Naval Post-Graduate School
- Syracuse University
- George Mason University

The expenditure of effort with respect to these organizations took on many forms:

- Sharing of technical ideas with respect to solving fundamental software engineering problems.
- With respect to the Andersen organization, collaborating on functional and technical objectives relative to commercial CASE tool efforts within FOUNDATION and AC Design.
- Supporting parallel research into KBSA/ADM topics by Syracuse University, Naval Post-Graduate School, and George Mason University.

Additionally, CSTaR has been an active participant in the last three Knowledge-based Software Engineering (KBSE) Conferences. Our work in this area has resulted in many papers, significant among them is Dr. Kevin Benner's work "Addressing Complexity, Coordination, and Automation in Software Development with the KBSA/ADM" (see Appendix A for a complete copy of this influential paper.) Andersen Consulting's participation in these conferences has resulted in a variety of KBSA/ADM program related publications:

Benner, K. M, Addressing Complexity, Coordination, and Automation in Software Development with the KBSA/ADM. Proceedings of the Eleventh Annual Knowledge-Based Software Assistant Conference, Syracuse, NY, September 1996.

Sparks, S, Benner, K; Faris, C. Managing Object-Oriented Framework Reuse. IEEE Computer, September 1996.

Benner, K. M., Tradeoffs in Packaging Reusable Assets. Proceedings of the Seventh Workshop on Software Reuse, St. Charles, Illinois, August 1995, (also available in http://www.umcs.maine.edu/-ftp/wisr/wisr.html).

Benner, K. M., WISR Working Group Summary: Software Reuse in A Business Environment--A Case Study. Workshop on Software Reuse, St. Charles, Illinois, August 1995. Published at WISR home page http://www.umcs.maine.edu/-ftp/wisr/wisr7/orgwg/orgwg.html.

Kim, J. J. and K. M. Benner, A Design Patterns Experience: Lessons Learned and Tool Support. Proceeding of The Pattern Languages of Object-Oriented Programs Workshop at the Ninth European Conference on Object-Oriented Programming (ECOOP '95); Aarhus, Denmark; August 7-11, 1995.

Naidu, E. B., and K. Miriyala, Space Creation and Compaction in Dynamic Diagramming. The International Workshop on CASE, 1995.

Sasso, W. and K. M. Benner, An Empirical Evaluation of KBSA Technology. Proceedings of the 11th Annual Knowledge-Based Software Engineering Conference, Boston, MA, November 1995.

Sasso, W. C. and K. M. Benner, An Empirical Evaluation of KBSA Technology. Proceedings of The Tenth Knowledge-Based Software Engineering Conference, Boston, MA, November 1995. IEEE Computer Society Press.

Sasso, W., Empirical Evaluation of KBSA Technology. Rome Laboratory, March 1995.

Luo, J., and K. Miriyala, A Practical Approach to Static Node Positioning. Proceedings of DIMACS International Workshop on Graph Drawing 1994. Lecture Notes in Computer Science, Vol. 894, pp. 436-443.

Benner, K. M., The ARIES Simulation Component (ASC). The Eighth Knowledge-Based Software Engineering Conference, Chicago, Illinois, September 1993.

Sasso, W. and M. DeBellis, Plan-Based Guidance for Knowledge-Based Software Engineering. Proceedings of the Software Engineering and Knowledge Engineering Conference, San Francisco, CA, June 1993.

Sasso, W., M. DeBellis, S. Bhat, O. Rambow and K. Miriyala, KBSA Concept Demo: Final Report. Technical Report RL-TR-93-38: U. S. Air Force Rome Laboratory, April 1993.

Sasso, W. and M. DeBellis, Plan-Based Guidance for Knowledge-Based Software Engineering. Short Papers of the 14th International Conference on Software Engineering (abbreviated form), Melbourne, Australia, May 1992.

Sasso, W., M. DeBellis and G. Cabral, Directions for Future KBSA Research. Proceedings of the Sixth Annual Knowledge-Based Software Engineering Conference, Syracuse, NY, September 1991.

Sasso, W., Motivating Adoption of KBSA: Issues, Arguments, and Strategies. Proceedings of the Sixth Annual Knowledge-Based Software Engineering Conference, Syracuse, NY, September 1991.

Sasso, W., Encouraging the Adoption of KBSE Technology: What Needs to Happen First? (editor, panel description). Proceedings of the Sixth Annual Knowledge-Based Software Engineering Conference, Syracuse, NY, September 1991.

Sasso, W. and M. DeBellis, A Software Development Process Model for the KBSA Concept Demonstration System. Proceedings of the Fifth Annual Knowledge-Based Software Assistant Conference, Syracuse, NY, September 1990.

Sasso, W., Empirical Study of Re-engineering Behavior: Design Recovery by Experienced Professionals. Software Engineering: Tools, Techniques, Practice 1(1), March 1990.

Benner, K. M., Simulation in Support of Specification Validation. Fifth Annual Knowledge-Based Software Assistant Conference, Syracuse, NY, September 1990.

Clearly from these interactions, the KBSA/ADM project has been influential.

# Conclusion

The KBSA/ADM project conducted by Andersen Consulting has been insightful and fruitful. The results of the project team's efforts begin to implement and describe how KBSA/ADM technologies address the software development problems associated with complexity, automation, and coordination. The most innovative aspects of this work have been in the areas of contextual knowledge (i.e., discussion databases, and object linking), process support (i.e., personalized agendas and process enactment), evolution transformations (i.e., transformations which automate stereotypical changes to a model), and critics (i.e., integrating intelligent analysis with process enactment).

While significant progress has been made, there are many areas for additional research and expansion on the work presented here.

- What is the appropriate development process for using the ADM? Current development processes have been constrained by the limits of the available tools and the discipline of the developers using them. More powerful and more intelligent tools should shift the balance of responsibilities and make new processes feasible which may not have been possible in traditional development environments.

- How effective is the synergy between the technologies described and implemented in this project? Have the technologies been integrated so as to support developers building complex systems or do developers find themselves trying to span awkwardly integrated tools and technologies?

- What might the effects of new implementation languages (e.g. JAVA) and the World-wide Web have on a KBSA/ADM implementation? As the convergence of networks, cheaper computers, and ever smarter systems spread across heterogeneous computing platforms, we might expect ever greater levels of software development collaboration at lower cost.

- What might a critic architecture look like and what is a full elicitation of critics? An organization's needs in terms of the number and types of critics required cannot be known a priori. A critic architecture, capable of quickly and easily capturing the "rules and regulations" by which a critic performs its analysis must be created.

These questions and others remain for researchers and entrepreneurs to explore using the KBSA/ADM technical architecture and existing tool foundation.

### Acknowledgments

The KBSA/ADM is the result of the work of many people without whom it would not have been possible. These people include past and present members of the KBSA/ADM development team: Kevin Benner (Co-Principle Investigator), Steve Sparks (Co-Principle Investigator), Chris Faris (Project Manger), Dave Gaffaney (IPSE), Jung Kim (ALE), Frank Luo (RASE), Enaganti B. Naidu (SAM), Bill Sasso (Contract Manager), George Ding (RASE), Mike Mikhail (ALE), Steve Killian (ALE, NT Port), Ilango Radhakrishnan (Remote Operations, SAM), Mike DeBellis

(Principle Investigator), Jim Coker (ARGO), Xiangyang Shen (Technical Architecture), and Sudin Bhat (Technical Architecture).

Appendix 1

# Addressing Complexity, Coordination, and Automation in Software Development with the KBSA/ADM

Kevin M. Benner
Center for Strategic Technology Research
Andersen Consulting
kbenner@cstar.ac.com

## Abstract

*This paper will describe how the Knowledge-Based Software Assistant / Advanced Development Model brings together technologies from the KBSE domain along with more traditional software engineering practices in order to address the pervasive software development problems associated with complexity, coordination, and automation. This paper describes how the KBSA/ADM realizes and refines the vision of Rome Laboratory's Knowledge-Based Software Assistant research program. The most innovative aspects of this work have been in the areas of contextual knowledge (i.e., design history, discussion databases, and object linking), process support (i.e., personalized agendas and process enactment), evolution transformations (i.e., transformations which automate stereotypical changes to a model), and critics (i.e., integrating intelligent analysis with process enactment).*

## 1. Introduction

The Knowledge-Based Software Assistant (KBSA) program is a Rome Laboratory funded effort to provide automated assistance to individuals and teams of software developers spanning the entire life-cycle of large software projects. The vision for this program was first articulated in "Report on a Knowledge-Based Software Assistant"[Gr93] in 1983. Since then, Rome Laboratory has funded a variety of research efforts which have made significant steps toward realizing and refining this vision ([Sm93], [Jo93], and [De92] among several others). Today, Andersen Consulting's Center for Strategic Technology Research (CSTaR) is building a field prototype of a KBSA called the Knowledge-Based Software Assistant / Advanced Development Model (or ADM for short). The ADM embodies many of the results of preceding research efforts, as well as refines the KBSA vision as it relates to addressing today's software development problems. The goal of this paper is to answer the question, "How does the ADM integrate KBSA technologies to address pervasive problems in software development?"

This paper is organized as follows: Section 2 will describe the problems associated with building and evolving large software systems. Section 3 will describe the ADM from a functional perspective. Section 4 will describe the details of what the ADM does---how the ADM manages complexity, automates what is understood, and supports coordination. Finally, section 5 will conclude with accomplishments and open issues.

## 2. The Problem

There are various problems associated with building and evolving large systems. Some of the most significant problems fall under the headings of complexity, collaboration, and automation. The problems under each of these topics are described in the following:

### 2.1. Complexity

As Brooks suggests, "The complexity of software is an essential property, not an accidental one"[Br87]. Booch elaborates on this point by saying, "We observe that this inherent complexity derives from four elements: the complexity of the problem domain, the difficulty of managing the development process, the flexibility possible through software, and the problems of characterizing the behavior of discrete systems"[Bo94] Because of this, only a small number of developers ever really understand the entire system. This understanding, however, is critical for finding and resolving inconsistent requirements, developing a coherent design, factoring the total system into manageable pieces for development, and evolving the system when requirements change in unanticipated ways. Critical design decisions,

assumptions, and rationale are seldom documented and indexed. Project memory and knowledge transfer are particularly problematic when evolving deployed systems or utilizing reusable assets. This is further exacerbated by the loss of the original asset's developers. Even if original developers remain with the project, over time they will not be able to remember all pertinent facts about the system.

### 2.2. Automation

In any development there are inevitably routine tasks which are well understood but tedious to perform (e.g., making changes to a program or specification which are not localized to one spot; or running an analysis tool like Lint and then making the necessary changes to remove the reported errors and warnings). These tasks take time to perform and too often, because of their routine nature, result in additional, avoidable errors which also have to be addressed. In a small project, one lives with these inconveniences. In an iterative development, this sort of thing happens all the time. In large projects, the cost of performing these small tasks and fixing the resulting avoidable errors compound quickly.

### 2.3. Coordination

As teams get larger, coordination becomes more essential, but harder to achieve. No matter how well a system is factored, communication among teams and within teams is necessary. This is true because of dependencies between work products, tasks, and the people performing the tasks. In a typical case, knowledge of how the system should work is fragmented across different teams and people. Curtis, et al, refer to this as "the thin spread of application domain knowledge"[Cu88]. The difficulty in these communications is disseminating knowledge in a timely and understandable manner to interested parties. Too often this knowledge is represented as static work products which soon become out of date. A big part of enabling coordination is establishing for developers the proper context in which to understand work products so that they can use and evolve them properly.

At the highest level, solutions to the above problems are not found in any single technology, technique, or capability, but rather they are found in the synergies between them. Within the ADM, the unifying theme is the Assistant metaphor. In this metaphor, the assistant brings to bear appropriate solution technologies in a coherent way to aid the developer or a team of developers address the problem at hand. This paper will describe how the ADM brings together several technologies from the KBSE domain along with more traditional software engineering practices. The next section will provide an overview of the ADM by describing it from a functional perspective. The following section will dive into the component technologies and how they address these problems.

## 3. A Functional Description of the ADM

The focus of the ADM is to provide intelligent, process-driven support to a team of individuals who are developing and evolving object oriented software/systems. This effort has had two driving goals: (1) to provide a suite of intelligent, process-driven, integrated software development tools, and (2) to provide a tool construction and integration framework in which tools may be built and integrated. As a tool construction and integration framework, the ADM provides the necessary openness to allow deep process enactment and intelligent assistance within and between individual tools and their work products. The framework provides a foundation upon which to build future tools and assistants as their need becomes evident.

As a suite of software development tools, the ADM provides set of capabilities which demonstrate the utility of automated support across the entire development life-cycle for both individual developers and teams of developers. Automated support includes the ability to manage informal knowledge in either an unstructured manner (i.e., as hypertext) or in a more structured manner (i.e., typed nodes and links reflecting important relationships---a la REMAP [Ra92]. Where more formal models are needed, the ADM supports specification creation and evolution of object-oriented models in a manner similar to ARIES' support for multiple views and evolution transformations[Jo93]. When multiple models are necessary, the ADM either ensures consistency within and between models or points out the inconsistencies to the developer. A specification, when complete, may automatically be translated into C++. Automatic code generation allows the developer to perform both system development and long term maintenance at the specification level. Testing is supported by automatically producing checking code to ensure stated specification conditions are satisfied.

Evolvability of a software system is supported in a variety of ways. Extensive work product linking capabilities allows one to capture either automatically or manually the rich dependencies between work products and the design rationale behind the creation and modification of each work product. Analysis of work products and changes to them are accomplished

via intelligent critics which evaluate their quality and suggest ways to improve them. Adaptability is further supported by lifting developer's activities from the code level to the specification level. The ADM provides for automating or assisting in the creation and modification of specifications including: performing stereotypical changes, coordinating complex modifications, and incorporating and maintaining design patterns/clichés and design rationale within a specification.

The ADM is currently made up of three principal tools which are tightly integrated via the ADM Technical Architecture:

- The Argo Language Environment (ALE) supports graphical and textual viewing and evolving of object oriented specifications. Critics evaluate the quality of the specification and suggest ways to improve it. When a specification is compete ALE can generate C++ code for a system, subsystem, or component.
- The Integrated Process Support Environment (IPSE) guides and coordinates teams of (multiple) developers in their execution of various software development tasks. The focus of IPSE is on task management via personalized agendas and task automation via enactment of tasks from one's agenda. IPSE supports project-wide project planning via its integration with MS Project™ for project plan development and modification.
- The Requirement Acquisition Support Environment (RASE) creates and manages informal documents, text, and pictures typical to requirements acquisition and analysis. In particular, it provides a hypermedia tool for capturing less structured, informal notations like natural language text, existing documents, and multimedia clips. RASE also provides a semi-structured representation based on typed nodes and links (a la REMAP) in which one can capture design rationale as part of a structured, multi-party, on-line discussion. While the focus of RASE has been on requirement acquisition, its capabilities are accessible throughout all life-cycle phases to capture informal knowledge whenever necessary. For example, REMAP discussions are often associated with deliverables and used as design discussion databases. In this way, discussions about alternative modeling decisions can be captured and directly related to the appropriate parts of the specification.

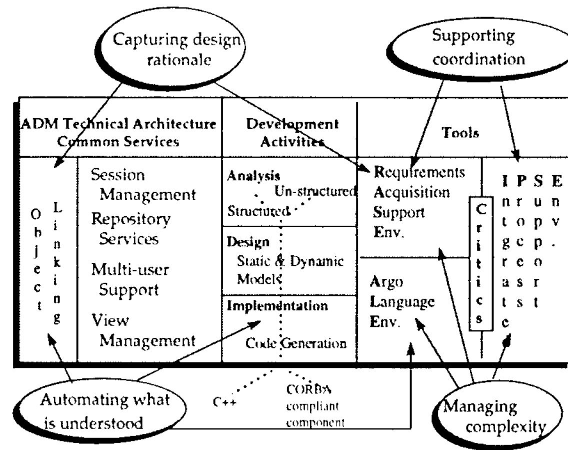



Fig. 6: ADM Common Services, Tools, and Development Activities

## 4. The Solution

While there are many technologies, techniques, and capabilities which makeup the ADM, this paper will group them under three broad categories: managing complexity, automating what is understood, and supporting coordination.

### 4.1. Manage Complexity

Management of complexity is a long recognized issue for large-system developers. The ADM addresses this issue by a combination of the following: (1) support for appropriate notations in which to capture and evolve software development work products, (2) mechanisms to maintain consistency between interdependent work products, (3) mechanisms to capture and make accessible contextual knowledge about why work products are the way they are, and (4) explicit project task support to formalize and enact the process of building a software system.

#### 4.1.1. Appropriate Notations

Appropriate notations provide the necessary underlying representations to capture desired information and present it for viewing and modification in an understandable and useful fashion. Within the ADM, underlying representations are expressed as metamodels. Presentations are described as graphical or textual projection defined in terms of the underlying metamodel. This is similar to the approach pioneered in ARIES[Jo92].

In the three tools which currently constitute the ADM, we see several examples of distinct representations and presentations. ALE's representation is the metamodel of the object oriented specification language that it uses---Argo. This

model includes concepts like class, attribute, relation, operation, and package (a unit of encapsulation analogous to folder or module). From this metamodel, ALE defines several presentations including a class diagram (a diagram showing classes and the relationships between them), a package diagram (a diagram showing import relationships between packages), and an Argo view (a view showing of a textual projection of our object-oriented specification language)

RASE's representation is an extension of the REMAP metamodel. Its extensions include more general concepts necessary for term-definition pairs found in the Term Dictionary, as well as hypernodes and hyperlinks found in the HyperText Editor. RASE supports one view for each of its component tools. In the HyperText Editor, hypernodes which are connected by hyperlinks are visualized in an outline format (a la the outline view in MS Word™). Within a hypernode, additional hyperlinks are viewed as "hot spots" which can be navigated to other hypernodes. In the Term Dictionary, terms are related to their definitions and then via object links associated with a REMAP term discussion. A term discussion is an instance of a REMAP discussion which, instead of focusing on requirement elicitation and refinement, is focused on deciding what concepts should be terms.

IPSE's metamodel includes the concepts of task, plan, resource, deliverable, dependency, and resolution. Its principal view is a personalized agenda of tasks which have been assigned to an individual. Views of the entire project are realized by maintaining an isomorphic representation in MS Project™ where more traditional project management views are supported.

Looking at the preceding metamodels, it is important to note the varying degrees of formality which are present. The Argo metamodel (like most specification languages) has a precise semantics. Several checkers have been developed to enforce various sets of semantic rules. The RASE metamodel has a much looser semantics. It allows users to classify information by using its typed nodes and links. This imposes a structure on the information, but does not impose a specific semantics which can be checked and enforced. At the informal extreme, HyperText is almost completely free form. It captures blocks of text as hypernodes connected via traverseable links.

Support for metamodels of varying formality is an essential strength of the ADM. Experience with systems which are all formal or all informal have demonstrated that reliance on one extreme to the exclusion of the other is inadequate.

Within the ADM all of its metamodel concepts are derived from a base concept which provides persistence, versioning, distribution, design histories, and object linking (i.e., a mechanism for hyperlinking between concepts in the ADM).

The metamodels, views, and tools described above are not meant to be definitive of the best set of metamodels, views, and tools. Rather, they are meant to convey the kinds of metamodels, views, and tools which can be defined within the ADM.

#### 4.1.2. Multiple Models and Maintaining Consistency

One of the principal differences between the current KBSA vision and that described in the original KBSA report is the presence of multiple models. In the original report, only one model is built. This is the system specification. It captures all the essential features of the system to be built. Development is the process of gathering additional information and evolving the specification of the initial model into the final specification from which an implementation may be generated.

The problem with this approach was that the single model became too complex to easily comprehend. ARIES [Jo92] addressed this problem by providing alternative views on the model based on projections of it. While very useful, it was insufficient for all of the ways people needed to see a system description in order to validate and evolve it. In particular, many of the simpler models developed early in the development process continue to play important roles later on. Evolution, which does not preserve these simpler models, makes comprehension and validation more difficult. Some people might think that these simpler models could be preserved and accessed as earlier versions of the more complex models. This would be an incorrect use of the concept of "version".

These simpler models are more than just abstractions of more detailed models. They are artifacts unto themselves which need to be preserved and evolved as necessary. One way to think about software development is as the creation of a series of models. Earlier models might be called requirements models. Latter models might be called implementation models. In between are models like conceptual models, functional models, design models, technical architecture models, etceteras.

Rather than transforming one model into the next, the ADM is more closely aligned with Rumbaugh's vision when he advocates "Each layer [model] captures design decisions made at that particular stage of the life-cycle of a particular system element"[Ru96]. Breaking the development into multiple models allows the developer to separate

concerns and express design decisions within the model in which it is most relevant and easily understood. This not only eases the development process, but also aids the understanding process of why things are the way they are.

In the ADM we are concerned with being able to create these models, capture the dependencies between these models, and mediate and enact the process by which these models are created and evolved. Managing complexity is more than just being able to differentiate between each of the above models. It is also being able to support the best ways to capture the information which makes up each of these models and maintain consistency between interdependent models.

As an example, consider the creation of a conceptual model. Inputs to this task include transcripts of interviews with stakeholders, user manuals and functional descriptions of legacy systems, white papers, etceteras. In the ADM, these are captured in the HyperText Editor. As requirements acquisition proceeds, discussions about conflicting requirements and elaboration of requirements are performed in REMAP. Eventually, common terminology needs to be identified by extracting terms from the HyperText Editor and creating entries in the Term Dictionary. Over time, terminology is consolidated and refined based on feedback from stakeholders. These terms eventually become the starting point for a more formal domain model to be expressed as Argo and built with ALE to reflect the classes, objects, relations, and behaviors of the conceptual model. The resulting conceptual model is a formal specification with traceability links to terms in the Term Dictionary and relevant portions of a REMAP requirements discussion on what should and should not be part of this model.

Throughout the above process, not only are work products being used to create subsequent work products, but dependencies are being created between the work products. A failing of many multi-model development approaches is that these dependencies are not maintained and the earlier work products typically become inconsistent. The ADM addresses this problem by creating and maintaining *object links* between dependent objects---even when they span models. Via this mechanism changes in one model result in changes in dependent models (more on this in the next section when talking about object linking). This sort of behavior is essential for iterative development methodologies where individual models are continuously modified. The failure to manage this problem has long been one of the principal shortcomings of traditional CASE tools.

### 4.1.3. Capturing Contextual Knowledge

Simply representing a model is seldom sufficient to be able to understand, maintain, evolve, or reuse it. It is necessary to understand why it is the way it is, as well as how it got that way. The "why" and "how" questions define the context in which a model exists. This context is often the informal knowledge which is in the heads of developers when creating a model. The ADM provides three mechanisms for capturing context knowledge: design histories, design discussions, and object linking.

**Design history** is a record of all the modifications which have happened to a work product. In the ADM, modifications are captured at the granularity of an editing gesture from the user interface. Each time a modification occurs, a record is created of the specific change. This record is then added to the detailed design histories of each of the objects it changes and is added to the design history of the current active project task.

When a developer is interested in how some object became the way it is (e.g., the class Inventory), he/she can view the design history of that object. When viewing the design history of an object, one can see the sequence of modifications which were made to it. Viewing an individual modification, the user can see what the specific change was and what other objects were changed at the same time. Following links from a history record, one can see in which project task the modification was made. For example, one may see that the Inventory class was created during the task Create Functional Model. Further inspection of the task's design history shows that the Inventory class was later renamed to Manufacturer's Inventory and made a specialization of a new, abstract class called Inventory.

It is via this lattice of objects, tasks, and modification records that a developer can see how objects have evolved. This record of what was changed, by who, and with which other objects is a starting point for understanding the contextual knowledge about a work product.

While the task pointer in a history record provides the project task context in which the modification was performed, it still leaves unanswered the more general question on why the change was performed. The following two mechanisms provide some support for addressing this problem.

The second mechanism for capturing contextual knowledge is a **design discussion**. A design discussion is a straight forward application of REMAP to document design discussions rather than requirements discussions. The same metamodel developed for the requirements domain (i.e., issues, positions, arguments, assumptions, decisions, etc.) is

valid for the design domain. In general, the ADM allows the developer to create new discussions as he/she sees fit. Typically, a discussion is associated with one or more work products. Controversial issues are recorded in the discussion and then related via object links to the pertinent portions of the work product.

The third mechanism for capturing contextual knowledge is **object linking**. Throughout the ADM, object links are used to show that concepts which are not physically close to each other are in fact related. For example, an object link can show the relationship between a hypertext node which captures a requirement to a class in an Argo specification which is responsible for satisfying this requirement. While the notion of object linking (or more commonly hyperlink) is not new, the ADM has gone to considerable lengths to maintain links in spite of objects being deleted (e.g., cause object link to point to old version only), modified (e.g., migrate pointers as objects evolve), and versioned (i.e., cause object links to migrate to latest version).

The semantics of a Basic object link is fairly weak. Its presence between two concepts shows that two concepts are related. If either of the concepts are deleted, then the object link will also be deleted. If a new version of either the source or destination objects is created, the link follows (or migrates) to the latest version.

A variant of the Basic object link, the Fixed object link, will remain fixed on the current versioned object regardless of the existence of more recent versions.

A more interesting specialization of a Basic object link is a Depends_on object link. If the target object of a Depends_on object link is deleted, the ADM notices this as an inconsistency and proposes alternative ways of resolving the inconsistency. These options include deleting the object link and the source object, mediating the selection of a new target, or creating a surrogate target to be replaced later. The basic goal is to ensure that the consistency rules for Depends_on object links are maintained.

#### 4.1.4. Project Task Decomposition and Enactment

This element for managing complexity focuses on the process dimension of software development. This includes both the macro and micro processes. Macro processes are defined in terms of tasks from a project management perspective. The process model for a

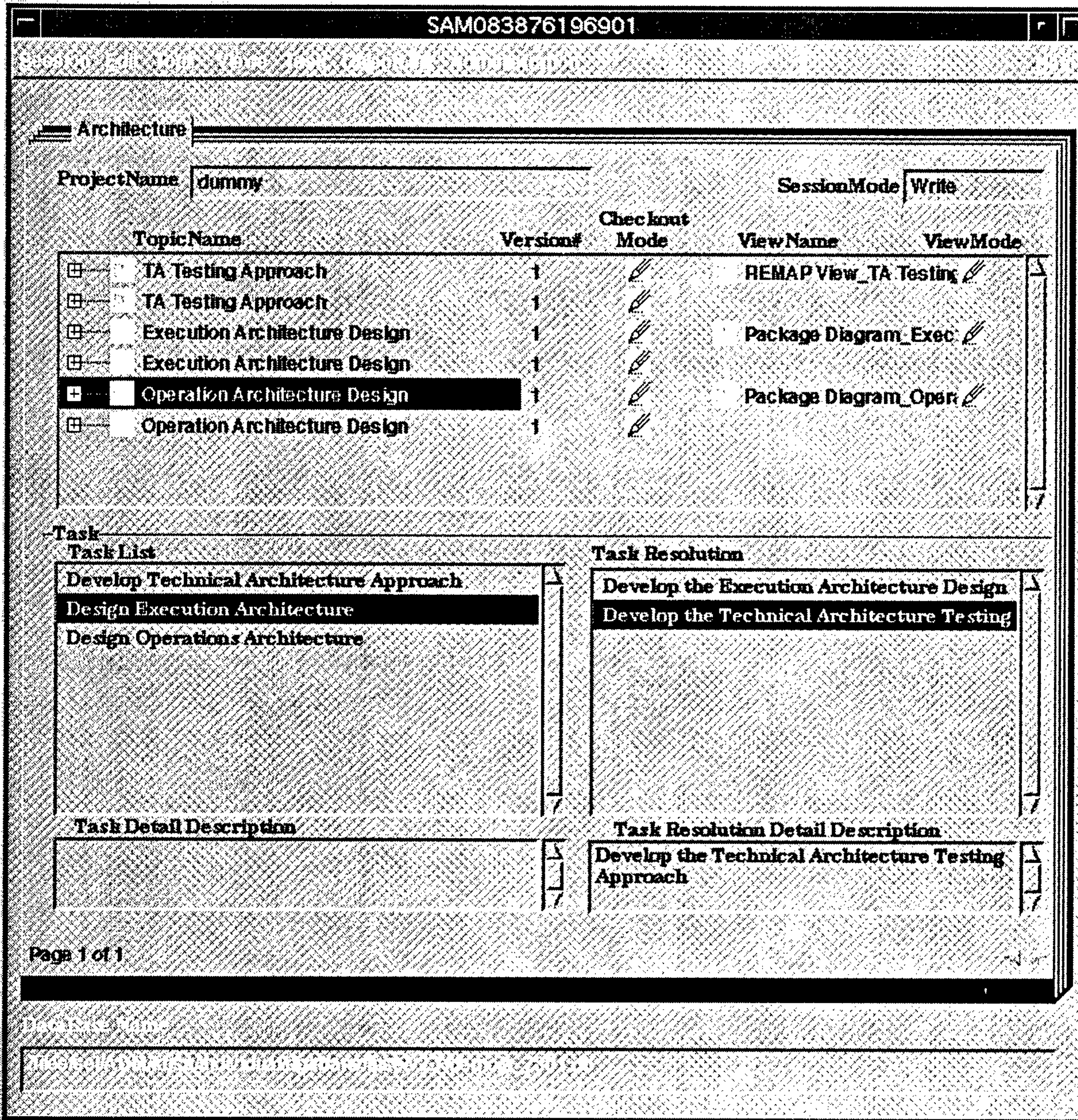



Fig. 7: ADM Session and Agenda Manager

macro process includes: resource allocation, task dependencies, and task status. Micro processes are focused on the detailed tasks performed by a single developer in his/her session. A Critique Resolution Task is a micro process task. It is created by a critic. It typically entails performing some very specific task to resolve a problem discovered by a critic, and is discussed further below.

Each developer has an agenda for each of his/her sessions. Figure 2 shows which work objects have been checked out into the session "Architecture", as well as the outstanding tasks to be performed in this session. For the selected task, "Design Execution Architecture", two alternative resolutions are shown.

Within the ADM, macro process tasks are defined using MS Project™. In general, the ADM uses MS Project for defining a project's tasks, their dependencies, and the allocation of resources. When imported into the ADM, MS Project tasks are converted into IPSE's task representation. IPSE, which is responsible for task management in the ADM, then distributes tasks to the personal agendas

of the developers assigned to them. IPSE synchronizes the IPSE and MS Project task models during import and export.

The developer can see what tasks have been assigned to him/her, as well as upon which tasks his/her tasks are dependent. Though this may seem simple, it is a significant step toward reducing the process complexity from the perspective of an individual developer. In a following section, we will see how things are assisted even more by supporting enactment of tasks on one's agenda by providing automation support for the supplied resolutions.

## 4.2. Automate What Is Understood

The place where the knowledge-based (KB) part of KBSA is most evident in the ADM is in the area of automation. Automation can be broken down into four broad categories: Evolution Transformations, Critics, Process Enactment, and Code Generation.

### 4.2.1. Evolution Transformations

Evolution Transformations (ETs) are editing operations which make complete semantic changes to a model. Their intent is to formalize as a single operation stereotypical editing operations which are made up of several other editing operations. This work is based on the ET work done in ARIES[Jo91]. The ADM extends this work by defining ETs which manipulate object-oriented specifications, as well as create and maintain object links between dependent concepts. The following are some example ETs:

- *Transform a relationship into a class:* When evolving simple models into more complex models, a relationship between two classes must often be transformed into a class in order to model more complex information about the relationship. This ET replaces the original relationship with a class and two new relationships. The cardinality of the new relationships is set to be consistent with the original relationship. All object links to the original relationship are moved to the new class. Finally, accesses which traversed the original relationship are updated to include the additional level of indirection through the new class.
- *Encapsulate concepts in package:* When transforming a conceptual model into a functional model, information hiding becomes an issue. Packages provide the encapsulation to achieve information hiding. Encapsulating concepts in a package requires creating the appropriate import relationships between packages to preserve necessary visibility between concepts.
- *Change inheritance to delegation:* In a conceptual model one may show two classes as being related via inheritance. In the functional model, one may decide that this results in a model which is too tightly coupled and thus would prefer to use delegation instead. This ET will remove the inheritance relationship, add a reference relationship, update all references to the previously inherited properties and operations by adding the extra level of indirection over the delegation relation, and update the constructors of both classes with some stub code.
- *Create a new model from an existing model:* One way to create a functional model is to evolve the conceptual model into the functional model. If you want to preserve the conceptual model while also creating the functional model you can make a copy of the conceptual model and then evolve the copy. This ET both creates the model and creates object links between the objects of the two models. These links are the basis for maintaining traceability and propagating changes between models.
- *Formalize dictionary term as Argo concept:* This ET creates an Argo concept of the same name as the dictionary term from which it is created. Additionally, it creates a traceability link between the two concepts.

The advantages of ETs include: increased productivity over performing several smaller editing operations for the same semantic effect, reduced opportunity for errors which could be caused by forgetting to do one of the component edits or performing it incorrectly, and more accurate reflection of the users intent in the design history by capturing modifications at the larger granularity of ETs.

### 4.2.2. Process Enactment

Earlier we talked about how the project model represented tasks and their interdependencies. Simply representing them was a significant step toward understanding and managing the complexity of the software development process. Several researcher (including [Hu89] and [Ka88] have taken this concept further by monitoring process execution for correctness and automating ("enacting") selected process steps where possible. The ADM extends this work by supporting a broader range of enactments spanning both micro and macro process steps.

There are two types of tasks which can be in one's agenda and are enactable: Project Tasks (part of a macro process) and Critique Resolution Tasks (part of a micro process).

- *Project Task:* When the developer *begins* a Project Task (e.g., create static model of subsystem xyz) by selecting it from his/her

agenda to enact, IPSE automatically (1) ensures that the entry criteria are satisfied, (2) collects the appropriate work products for the task, (3) launches the tool(s) necessary for the task, and (4) updates the project task's status to "in_progress". When the developer *completes* the project task, IPSE automatically (1) checks that the exit criteria are satisfied. If the exit criteria are acceptable, IPSE (2) returns the work products to the repository, (3) creates a new version of modified work products, (4) creates object links between work products and the project task for traceability purposes, and (5) updates the project task's status to "completed".

- *Critique Resolution Task:* When the developer selects a Critique Resolution Task from his/her agenda, he/she is presented with a description of a problem which was found by some critic. Critics evaluate work products and report their findings by posting tasks to a developer's agenda. These critique resolution tasks point out particular problems or shortcomings in work products. Unlike typical analyzers, ADM's critics are intelligent in that they often include—as a part of their critique—specific, alternative means of fixing or improving the work product. The analyst may select the desired alternative, at which point the critic automatically or interactively makes the selected improvement. Alternatively, the developer may ignore the advice and do something completely different. In the latter case, the task may be deferred and/or revalidated. More information on critics will be presented in the next section.

#### 4.2.3. Critics

Critics are the best manifestation of an assistant in the ADM. In general, they are analyzers which both check models for desired properties and kibitz on how to fix the model should the desired property not be satisfied. This work draws most heavily on Fischer's critics work whose goal was to automate the mundane and help bring to bear knowledge at the appropriate time and place when the developer needed it [Fi91]. The principal innovation of the ADM in this area is the integration of critics with IPSE. IPSE's agenda allows any critic to post a potential task or problem and its potential resolutions on the developer's agenda. The developer may then work on which every problem he/she is ready to work on.

As an example, consider the type checker/linker---a well understood analyzer. The ADM extends the traditional functionality of this analyzer to include suggesting ways to fix the errors it finds. When the type checker/linker discovers a dangling reference it also generates several resolutions each of which can fix this problem. In this case, the generated resolutions are (1) rename the reference, (2) define the referenced concept based on the reference, and (3) import a package into the current package so that the referenced concept is visible.

A resolution is basically a packaging of the evolution transformation which will make the desired modification so as to remove the error. Additionally, it includes some subset of its parameters already bound based on the context in which the error was discovered. For example, in the second resolution above, if the referenced concept is a class, a create class ET will be selected. The name parameter will be bound to the name of the referenced concept and the package in which to define the concept will be bound to the current package.

Problems and resolutions are presented to the user as Critique Resolution Tasks (CRTs) which when created are placed on the users task agenda. The user upon selecting a CRT is presented both with a description of the problem and the enumeration of the alternative resolutions. The user may then select the desired resolution which results in executing the associated ET.

CRTs are managed by the critic which created them. If a CRT is no longer valid (e.g., the model is modified to remove the error), the critic must tell IPSE to remove it from the agenda.

Currently the ADM has implemented three critics: a type checker/linker, an Argo semantics critic, and an object-oriented style critic.

#### 4.2.4. Code Generation

ADM code generation translates an Argo specification into C++. Toward this end, ALE has been successful in several ways. The principal success is preserving the semantics of Argo in the generated C++. In particular, Argo has a fairly rich data model which the generated code enforces. Specific features include: maintaining referential integrity on relationships, enforcing cardinality constraints on attributes and relations, creating appropriate destructors and copy constructors to realize the desired semantics of composition relations (e.g., cascade delete and deep copy over composition relations), and creating initializers for all properties. With respect to the dynamic model, preconditions, postconditions, and class invariants can be automatically inserted into the generated code to aide in testing and runtime verification of the specification.

Given KBSA's lineage back to automatic programming, code generation has always been considered one of the more important aspects of the ADM. As we have explored technology transfer opportunities within Andersen Consulting, we have

consistently found that code generation was of little interest to the organizations we have targeted. Basically, code generation is perceived as being of limited value. Coding at the unit level is cheap. Code generation may minimize the need for unit testing, but it does not address assembly test, system test, product test, or acceptance testing. The complexity is in describing all of the business process to be covered, not in actually coding up the individual business processes.

The problems with code generation as applied in a business setting are four-fold: (1) How to integrate with commercial COTS products (e.g., GUI builders, relational database management systems, object-oriented database management systems, and middleware of all sorts)? These products often have their own proprietary high level languages and have their own life-cycle of versions to deal with. (2) Will the a set of products ever be stable enough that the code generator can be used enough times to justify the cost of modifying it for the latest version of these COTS products? (3) Are the skills necessary to build and maintain code generators consistent with the skills available to a typical engagement (i.e., project team)? And (4) how much does adding generation technology to a development environment add to the project's complexity and associated risk?

A common request of front line developers within Andersen has been, "Get me through analysis and design, and then provide reliable pointers to where specific concepts are reflected in the code." In this context, code refers to either source code files or other source file (i.e., files which contain the proprietary high level language of the advanced COTS product). With appropriate object links, they could then do coarse grained impact analysis when requirements or COTS products change and thus be in a better position to manage when and where to change the implementation.

## 4.3. Support Coordination

A primary shortcoming of much work in the KBSE arena is that it has not addressed the issues associated with teams of developers. The ADM addresses this by providing a client/server development environment in which multiple people can work simultaneously. Within this environment the ADM provides repository management, task management, and multi-party discussions.

### 4.3.1. Repository Management

At the core of this environment is a centralized repository from which developers working on their own workstation can check-out work products and perform development activities in the context of a long transaction model. The repository supports both version and configuration control. Interaction with the repository is mediated by a session metaphor. When a developer wants to view or edit some work product, he/she checks the work product out of the repository for either read or write. The repository enforces a single writer, multiple reader protocol.

When a new version of a work product is created, the ADM uses the agenda mechanism of IPSE to inform all developers using older versions that the work product currently in their session is no longer the latest version. The notification appears as a critic resolution task and offers as a resolution to check-in the current work product and check-out the latest version. By using the agenda mechanism, the ADM gives the developer the flexibility of knowing as soon as possible that he/she is working with an older version, but allows the developer to delay integration of the new version until he/she is ready (maybe after finishing the task he/she is working on).

The versioning system also facilitates exploration of the design space by allowing alternative versions of a work product to be created. Central to managing alternative versions is the use of discussion databases (i.e., a REMAP discussion) in which developers can record why alternatives are being created. Object links are used to connect the discussion to the appropriate portions of the designs (in either version). In this way, the differences between versions can be highlighted and explained. When one version is finally selected over another, the rationale for the decisions can be captured as the conclusion of the discussion. The capture of this sort of information is central to supporting coordination and collaboration between multiple developers. This is another example of capturing contextual knowledge which would normally remain in the heads of a small number of developers who were directly involved in the decision. It is now available to other people on the team. As team personnel turns over, this will become even more important.

### 4.3.2. Task Management

As was described in earlier sections, the ADM formalizes and manages the Project Tasks of a development effort. From a coordination perspective, this formalization and management allows the ADM to identify and mediate resource contentions which can arise within a team of developers. The most obvious contention is clashes over work products. Because IPSE understands the dependencies between tasks it can moderate who should have priority with a given resource. Additionally, when a developer is denied write access to a work product, IPSE can inform the

developer who has the work product and what project task they are performing.

A more subtle contention occurs when developers begin using work products which they believe are released, but which are really still under development. IPSE manages this via both explicit status attributes on work products and formalized exit criteria on work products. In the latter case, when a developer says a work product is done, IPSE runs the exit criteria checks to ensure they are satisfied. In this way, when a developer goes to use a work product developed by someone else, they understand exactly what its state is and thus will use it correctly. That is, one may still want to use a work product even though it is still under development, but now with the knowledge on how stable it is likely to be.

#### 4.3.3. Multi-party, Structured Discussions

Previous sections described how REMAP discussions can be used to capture both requirements and design discussions. Because these discussions support multiple participants, they make a significant contribution toward supporting coordination and collaboration between competing interests as a consensus is striven for.

A criticism of REMAP discussions is that they are a mechanism outside of the normal process flow of a development team. Most significantly, meetings---where many critical requirements and design discussions and decisions occur---are outside the system and thus are a source of lost contextual knowledge. The response by the ADM is more methodological than technological. Where are the meeting notes captured?

It is certainly the case that meetings are critical for high band width discussions. But if notes are not recorded of what alternatives where considered and what decisions were made, then as we all know from personal experience, important details will be lost and the meeting will have to be held again (at least a short one to reconstruct the "what" and "why"). REMAP should be used to address this problem by using it to summarize the meeting after the fact. The advantage of this approach is that participants of the meeting will see the record and have the opportunity to correct the record. If desired, the discussion can be continued within REMAP. Of equal importance, people who were not part of the meeting have an opportunity to see what was discussed and add their input if necessary. REMAP provides a significant advantage over straight textual notes because it imposes a specific structure on the discussion which allows readers to quickly track the high level issues and then selectively dive down into greater detail as desired.

## 5. Conclusion

This paper has described how the ADM incorporates KBSA technology to address the software development problems associated with complexity, automation, and coordination. The most innovative aspects of this work have been in the areas of contextual knowledge (i.e., design history, discussion databases, and object linking), process support (i.e., personalized agendas and process enactment), evolution transformations (i.e., transformations which automate stereotypical changes to a model), and critics (i.e., integrating intelligent analysis with process enactment).

While significant progress has been made there are still several open issues. What is the appropriate development process for using the ADM? Current development processes have been constrained by the limits of the available tools and the discipline of the developers using them. More powerful and more intelligent tools should shift the balance of responsibilities and make new processes feasible which may not have been possible in traditional development environments.

How effective is the synergy between the technologies described in this paper? Have the technologies been integrated so as to support developers building complex systems or do developers find themselves trying to span awkwardly integrated tools and technologies?

Past KBSA and KBSE efforts have suffered from usability problems. Andersen's ADM precursor (the KBSA Concept Demo), while an excellent demonstration system, lacked many of the necessary capabilities a development team had to have. The ADM team has actively attempted to provide full coverage for the types of tasks developers must have in a development environment. In one case it meant integrating a COTS tool (i.e., MS Project™). In other cases, it meant building within our environment less sexy, but critical capabilities which are common in CASE tools (e.g., support for multiple developers). The result has been a development suite of tools which do span the full life-cycle and demonstrate the application of knowledge based techniques to address important software development problems.

Answers to the above questions can only be addressed by empirical evaluation. This is our next task within the ADM Project. Empirical evaluation of the ADM begins January 1997.

## Acknowledgments

The ADM is the result of the work of many people without whom it would not have been possible. These people include past and present members of the ADM development team: Steve

Sparks (Co-Principle Investigator), Chris Faris (Project Manger), Dave Gaffaney, Jung Kim, Frank Luo, Enaganti B. Naidu, Bill Sasso, George Ding, Steve Killian, Ilango Radhakrishnan, Mike DeBellis, Jim Coker, Xiangyang Shen, and Sudin Bhat.

This research is supported by Rome Laboratory of the Air Force Materiel Command under contract F30602-93-C-0015.

Rome Laboratory and Andersen Consulting are interested in making the technology described in this paper more broadly available. Contact the author if you are interested in getting direct access to the ADM or the frameworks from which it was built.

## References

[Bo94] Booch, G., Object-Oriented Analysis and Design with Applications, 2nd edition, Benjamin/Cummings Publishing Company, 1994, p. 5.

[Br87] Brooks, F., "No Silver Bullet: Essence and Accidents of Software Engineering," IEEE Computer vol. 20(4), April 1987.

[Cu88] Curtis, B., Krasner, H., Iscoe, N., "A Field Study of the Software Design Process for Large Systems," Communications of the ACM, Vol. 31(11), November 1988.

[De92] DeBellis, M., Miriyala, K., Bhat, S., Sasso, W., and Rambow, O., "Final Report: Knowledge-Based Software Assistant Concept Demonstration System," CDRL A007, Government contract F30602-89-C-0160.

[Fi92] Fischer, Lemke, Mastaglio, Morch, "Critics: An Emerging Approach to Knowledge-Based Human Computer Interaction," International Journal of Man-Machine Studies, 35(5), pp. 695-721, 1991.

[Gr83] Green, Luckham, Balzer, Cheatham, and Rich, "Report on a Knowledge-Based Software Assistant," RADC TR 83-195, Rome Laboratory, 1983.

[Hu89] Huff, K. "Plan-based intelligent Assistance: An Approach to Supporting the Software Development Process," doctoral dissertation, Dept. of Computer Science and Information Science, Univ. of Massachusetts, Sept. 1989.

[Jo91] Johnson, W.L., Feather, M., "Using Evolution Transformations to Construct Specifications," in Automating Software Design, edited by Lowry and McCartney, AAAI Press, 1991.

[Jo92] Johnson, W.L., Feather, M., Harris, D., "Representation and Presentation of Requirement Knowledge," IEEE Transactions on Software Engineering, Vol. 18(10), October 1992.

[Jo93] Johnson, W.L., Benner, K.M., and Harris, D.R., ``ARIES: Developing Formal Specifications from Informal Requirements,", IEEE Expert, Sept. 1993.

[Ka88] Kaiser, et al, " Intelligent Assistance for Software Development and Maintenance," IEEE Software, May, 1988.

[Ra92] Ramesh, B., "Supporting Systems Development by Capturing Deliberations During Requirements Engineering," IEEE Transactions on Software Engineering, Vol. 18(6), June 1992.

[Ru96] Rumbaugh, J., "Layered Additive Models: Design a process of recording decisions", Journal of Object Oriented Programming, Mar/Apr 1996.

[Sm93] Smith, D.R., Parra, E.A., "Transformational Approach to Transportation Scheduling", In proceedings of the 8th Annual Knowledge-Based Software Engineering Conference, Chicago, IL, Sept 20-23,

# Appendix 2

# RASE: an Integrated Requirements Acquisition Support Environment

**Junhui Luo and Kevin M. Benner**

Center for Strategic Technology Research
Andersen Consulting
3773 Willow Road
Northbrook, IL 60062 USA
+1 847 714 2453
{luo,kbenner}@cstar.ac.com

**ABSTRACT**

This paper describes the capabilities of the **R**equirements **A**cquisition **S**upport **E**nvironment (RASE). RASE supports the capture of informal requirements; the negotiation and consolidation of conflicting requirements; and the transformation of informal information into formal descriptions. Informal hypertext documents, requirements discussions, consolidated term dictionaries, and formal object oriented models are the results of these activities. In the context of both these processes and these work objects, RASE provides structured traceability links between work objects, as well as to their respective evolution history and rationale. The uniqueness and strength of RASE lies in the integration of a broad set of capabilities not found in any other single tool. Other tools have had limited impact on the requirements task because they have been too focused on only small number of activities rather than the broad set of activities necessary during requirements acquisition.

**Keywords**

Requirements acquisition, design rationale, structured discussion, evolution transformation, traceability.

**INTRODUCTION**

Traditional software engineering approaches treated systems requirements as a set of document---typically textual documents with diagrams. Little help was provided to maintain consistency between document nor to link these documents to down stream development work objects. Some approaches used only natural languages with informal semantics, some used semi-formal notations. These approaches tended to suffer from the following essential problems.

First, requirements do not exist as distinct things ready to be collected. In fact, requirements for complex systems need to be constructed as the result of an engineering process. Second, most human information exchange activities happen in the context of a large number of unsaid assumptions. As a result, direct capture of requirements and requirements discussion may fail to capture such information. Third, requirements rationale (i.e., why these requirements) needs to be captured, along with the requirements themselves. Fourth, requirements information is voluminous and must be properly structured to be useful for subsequent life-cycle activities.

In recent years there have been a number of research efforts to address the above problems, but these individual efforts have had limited impact on the requirements task because they have been too focused on one activity rather than the broad set of activities necessary during requirements acquisition. We contend that a requirements acquisition support environment must address *all* of these activities in order to be effective at alleviating the problems of today's software development projects. The uniqueness and strength of RASE lies in the synergy it creates by integrating a variety of requirements activities.

We have taken a transformational approach toward the requirements engineering process. The idea bears similarity to Johnson' and Feather's work [3] in the specification area. We view the requirements engineering process as the process of eliciting initial informal information and gradually transforming the informal information into formal representations. As being commonly observed [1, 4, 10], during this process we expect conflicting requirements to emerge and to be the subject of discussion until a common position is achieved. We provide multiple models to cover the full spectrum of informal and formal information representations. RASE provides a tool for the manipulation and viewing of each representation:

- Hyper text documents enhanced with multimedia objects, e.g., video, audio, and graphics are used to capture raw requirements information. A structured hypertext composition / navigation tool is used to provide traceability among requirements elements.

- A tool supporting the REMAP [1] model is provided for structured requirements and design discussions. Each discussion element, e.g., an issue or a position, in the REMAP tool is itself a hyper-document and thus can be linked to other design artifacts.

- RASE also has a Term Dictionary tool that is used to capture important project concepts and terms. The Term Dictionary tool is also built on top of the hyper document tool. Therefore, term definitions in the Term Dictionary can be hyper-linked to their uses and related design elements.

- Integration of these capabilities with other design and development tools in our environment, such as the Specification Language Diagrammer, supports full traceability and design rationale capture of life-cycle artifacts.

The rest of this paper is organized as follows. The next section describes the problems in requirements engineering that RASE addresses. The following section will describe the capabilities of RASE in detail. Next we briefly mention some implementation information of RASE. Finally, we conclude by comparing our work with related works, summarizing our contribution and pointing out future directions.

## PROBLEM STATEMENT

First, requirements do not exist as distinct things ready to be collected. In fact, requirements for complex systems need to be constructed as the result of an engineering process. There are various reasons that this is so. Initial requirements may only exist in people's mind as vague ideas, they may not be well thought of, can conflict with each other, and are incomplete. Or individuals with different social or organizational background may have different and conflicting project goals, and they may make various *implicit* assumptions that they assume to be true and do not actively communicate these default assumptions. Before proceeding to subsequent phases of development, these vague ideas need to be articulated, conflicts need to be resolved, a consistent set of requirements need to be arrived at, and the requirements need to be prioritized. Models, processes, and tools are necessary to assist in this process. Lack of such guidance and help results in ineffective and inefficient requirements acquisition activities.

Second, research in social sciences has demonstrated that many important human information exchange activities happen in the context of a large number of unsaid assumptions. In order to communicate, participants may need to make enormous amount of implicit assumptions about the social, organizational, and domain context where a conversation takes place. Such assumptions are necessary for participants to interpret and make sense of what is being spoken or written. Therefore, it may not be sufficient to just capture requirement statements alone, but also the underlying implicit assumptions and context required to understand and interpret each requirement statement.

Third, because requirements information can be voluminous, often times requirements documents become write-only. It is hard for project personnel to locate relevant information when the documents lack structure appropriate to support design and development activities and when there is minimal tool support.

Fourth, today's software systems are so complex that no individual understands an entire system. In fact, understanding of the system itself is often not sufficient. But rather, history and rationale about why things are the way they are also need to be understood. This understanding is critical when further decisions need to be made, or when an

existing system needs to be changed to meet new requirements. This kind of critical systems knowledge is often only understood by a few key project members and is kept in their mind or informal notes, rather than formally captured and made generally accessible. As time passes, these team members' memory may become vague. When they leave the project team, this knowledge is lost.

These problems compound. The net result is poor requirements quality, and consequently poor product quality. Of course, they can also lead to schedule and cost overrun, project failure or cancellation.

Our answer to these problems is to provide computer support for the broad set of requirements acquisition activities.

Using computer to capture the enormous amount of knowledge and contextual information of a software system greatly alleviates the burden on the human's part. Properly structured contextual information of the right granularity that is hyper-linked to life-cycle artifacts provides developers with convenient access to information that is relevant.

## RASE CORE FUNCTIONALITY

RASE takes a transformational approach toward requirements gathering. As we mentioned earlier, we view the requirements engineering process as the process of eliciting initial informal information and gradually transforming the informal information into formal representations. During this process we expect conflicting requirements to emerge and to be the subject of discussion until a common position is achieved. Two essential problems associated with today's large scale software development projects are related to the issues of complexity and automation. RASE addresses the complexity issue by providing notations and tools for properly structuring information; and addresses the automation issue by supporting high level semantic operations on information to relieve humans from performing mundane, tedious, but well-understood operations thus reducing the likelihood of errors that humans may make when performing such operations.

In this section we will describe in detail the capabilities provided by RASE.

### The RASE Requirements Acquisition Process Model

Initial systems requirements are gathered from multiple sources. They can be from previous systems operations manuals, interviews and meetings with user and other stakeholders, etc.. They are often expressed using written natural languages, drawings, or verbal communications. As a starting point, this information needs to be captured. RASE captures this information via hypertext documents which support textual, graphical, audio and video representations.

As the initial requirements are gathered from multiple stakeholders, they are usually ambiguous, inconsistent, and incomplete. They need to be analyzed and consolidated. One way to accomplish this is by adding formality to the representation. For examples, requirements ambiguity or inconsistency may stem from the fact that terminology is only loosely defined. These problems may be the result of different people using different terms to name the same concept, or using the same term to name different concepts. These problems can be addressed by defining a unified and standard terminology. Requirements can also be disambiguated by using formal notations, e.g., formal specification languages.

The requirements engineering community has long realized that this informal to formal transformation task is a complex process. It is widely agreed that in order to fully understand the outcome of this process, the rationale and the process history must be captured, along with the actual transformation. This rationale is usually the result of a multi-party discussion and negotiation process. [1, 2, 4, 10] proposed models and tools for structuring and capturing such processes.

RASE is designed specifically to support this view of requirements engineering. RASE provides a number of integrated tools to cover the spectrum of informal to formal systems descriptions, mechanisms for assisting the transition from informal to formal description, and mechanisms for capturing the rationale and history of the transitioning process. In the following, we will first describe each of the component RASE tools in more detail, then we will describe how the tools work together to provide the synergy for assisting the requirements capturing process, especially the transition from informal to formal description, capturing requirements rationale, and creating traceability links between requirements objects.

## Components of RASE

As we mentioned earlier, RASE consists of three integrated tools: the hyper document editor / browser, the REMAP discussion tool, and the Term Dictionary tool.

*The Hyper Document Editor and Browser*

The hyper document editor / browser supports capturing of informal information in English. The editor supports WYSIWYG style composition and point-and-click style navigation of hypertext. Figure 1 is a screen image of the hyper document editor / browser's user interface.

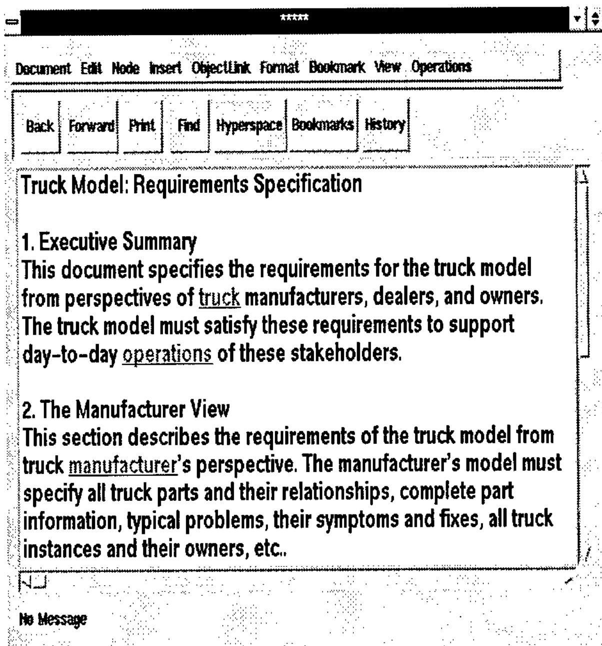



Figure 8: The hyper document editor / browser tool

Each hyper document is organized into a hierarchy of *hyper nodes*. The node hierarchy reflects the natural organization of a document into chapters, sections, subsections, etc.. Each hyper node contains text that may have embedded hyper-links. A hyper-link is a substring of the text in a node that is object-linked to another work object in the ADM repository, e.g., a piece of text in the same or another hyper document, a class, attribute, relationship or operation in a formal specification. A hyper node can also contain references to multimedia objects, i.e., pictures, audio or video files. These multimedia elements can be enacted via mouse-clicking, a la popular Web browsers.

Each hyper node can be assigned a string-valued type. An analyst can use this feature to mark a node as a requirement, an issue, etc.. As we will see later, this feature will also allow the exporting of marked node contents to other types of representations, for example, a REMAP discussion.

Organization of a hyper document into a hyper node hierarchy allows information management at a finer granularity. Hyper nodes are treated as first class objects in ADM. The editor / browser provides query engines that operate based on the node structure. For example, an analyst can specify a query to find all hyper nodes that are marked as requirement in a document. Nodes also provide unit of sharing. A new hyper document can be assembled from existing nodes in other documents.

*Bookmarks* can be created to mark any position inside the text of any node. Bookmarks provide the analyst with an easy way to remember important places in a document as well as serving as object-link target. For instance, an object link can be created that traverses from the concept "Elevator" in a formal specification to a bookmark in a hyper document where a textual description of an elevator can be found.

The hyper document editor tool also supports an outline view of a hyper document. In an outline view, only header (numbering and title) information of hyper nodes is shown. The hyper node hierarchy can be conveniently (re-)arranged in an outline view. The editor supports standard word processor functions such as text style change, cut/copy/paste, import / export, and print.

*The REMAP Discussion Tool*

REMAP [1] is an extension of the IBIS (Issue-Based Information System) [10] model. Figure 2 shows the REMAP discussion schema. The IBIS model is embedded as a sub-schema (indicated in dotted box). In the REMAP model, requirements acquisition task is viewed as a structured discussion and deliberation process. Given initial input requirements, issues concerning their fulfillment are raised. Positions that are possible resolutions of the issues are then proposed and arguments are generated that either support or object to positions. Eventually decisions are made that select certain proposed positions to resolve all issues. Finally, based on the decisions, refined or consolidated requirements are generated as output of the discussion process. The model enforces a structure on discussions, categorizes discussion

elements according to the role that they play, and identifies the semantic relationship among discussion elements. By following this model, the requirements acquisition process can be conducted more effectively and efficiently. For details of the REMAP model, refer to [1].

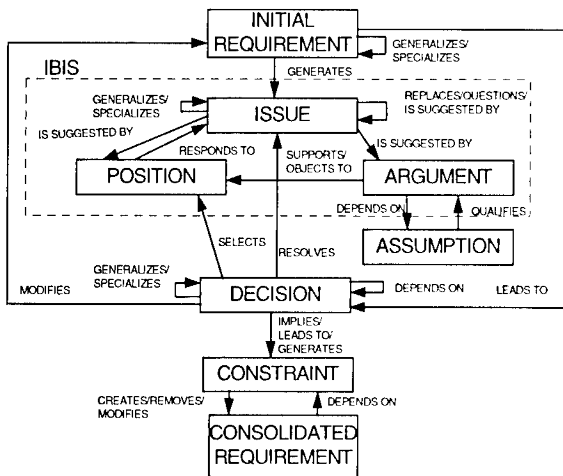



Figure 9. The REMAP model

The ADM REMAP discussion tool supports the REMAP model. The tool allows on-line multi-user requirements discussions. Elements of discussions are recorded in the database and are linked with relevant work objects. Figure 3 shows a screen image of the user interface of the ADM REMAP tool. The top part of the window graphically displays the structure of a discussion. Different types of discussion elements are shown using differently shaped and colored icons. Semantics of links between discussion elements are indicated using different colors. The bottom part of the window displays details of a discussion element. The details include the type of the discussion element (e.g., an issue or a position), a short name, the author (automatically captured by the tool), date created, and a detailed description. The detailed description field uses the same hypertext widget that the hyper document editor uses. Therefore, text in the detailed description field can contain hyperlinks to other ADM work objects, as well as multimedia objects. A user can use the palette items on the left side of the window to create discussion elements (issue, position, argument, etc.) and typed links. Each discussion element can also be assigned an *owner*. An owner can be different from the person who created the element. The owner of an issue is responsible for seeing the issue being eventually resolved.

By capturing such design deliberation process data in the same environment where the end products (work objects) are kept, important rationale and contextual information are kept and can be linked with the end products. By providing a spectrum of granularity of information (discussion elements, hyper node hierarchy), such links can be maintained at appropriate level as desired. This is important because the key to the understanding of a requirement or a piece of design is being able to locate its relevant contextual information, and only the relevant information. In the later section on evolution transformations, we will also describe how such traceability links can be automatically created.

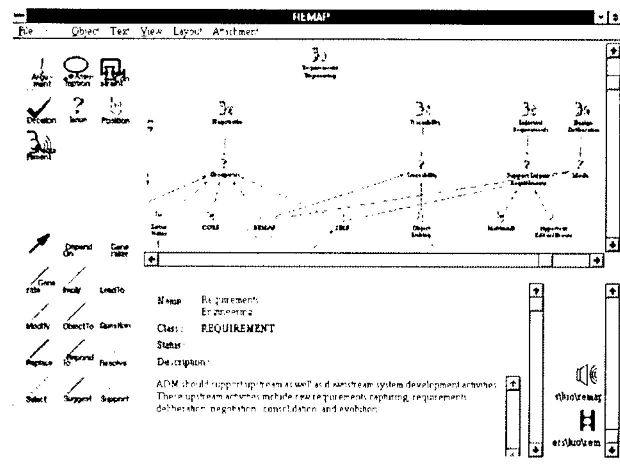



Figure 10. The REMAP tool user interface

Such deliberation discussion processes are part of the information refinement and transformation process. Inputs to a discussion are raw information, outputs from a discussion are processed information. In the case of a requirements discussion, inputs can be initial requirements from multiple stakeholders, which may contain ambiguities and conflicts with one another. After the stakeholders negotiate their initial requirements using the REMAP model and tool, the outputs from the discussion will be consolidated requirements that have resolved the ambiguities and conflicts.

*The Term Dictionary Tool*

In this section we describe the third tool in RASE, the Term Dictionary tool that provides a place where unified terminology can be defined.

It has long been recognized that as a large software project progresses through its phrases, it is important to keep track of the important terms that stakeholders

use to describe concepts relevant to the project. These concepts range from those in the original application domain (e.g., Inventory), to those created in the various phases of the project as results of the requirements engineering, design, and development activities (e.g., Inventory Control Module, User Interface Framework). There can be an enormous amount of such concepts and terms in a large project. These concepts and terms emerge and evolve as the project progresses. At the beginning of the requirements gathering stage, various concepts related to the system to be built are usually only vague ideas in people's mind. People use terms that are not well-defined to talk about the system. They may use the same term at different times to mean different things, or use different terms to mean the same thing. In a large scale project involving multiple stakeholders, individuals from different organizations are very likely to use different terms to describe the system to be built because they have different organizational and cultural background and the system to be built plays different roles in their respective organizations. But obviously it is essential that all stakeholders must come to talk in a unified "language" before they can achieve consensus on what is to be built. Data dictionaries in traditional CASE tools provided a facility for capturing data items in a project. However, in a real life large scale software project, there is strong evidence and tendency that these data dictionaries are not actively maintained, resulting in out-of-date, incomplete, and inconsistent data definition items. Thus data dictionaries in traditional CASE tools fell short of supporting the active maintenance of important project terms.

The RASE Term Dictionary tool addresses these problems by leveraging and synergizing the ADM hypertext engine and the Object-Linking mechanism to make the term maintenance effort easy. The Term Dictionary uses the hypertext engine as its front-end, thus allows the definition of a term to be hyper-linked with its uses through out the project database.

Figure 4 shows the user interface of the Term Dictionary tool. The Term Dictionary tool is built on top of the hyper document editor. Terms are alphabetically ordered in the dictionary. Through user interface menus a user can define and modify terms and their definitions. Because the term definition field is a hypertext widget, the term definition can be hyper-linked to places where the term is used, or linked to other related work objects. This pervasive use of typed object links makes it possible to ensure consistency of term usage across a project. This powerful object-linking capability distinguishes the RASE Term Dictionary tool from data dictionaries in traditional CASE tools.

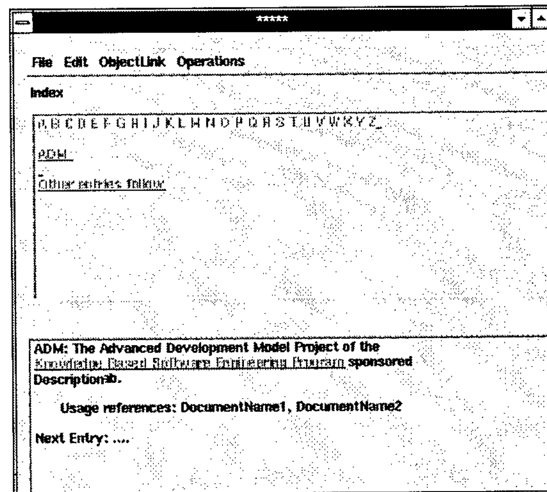



Figure 11. The Term Dictionary tool user interface

The Term Dictionary tool assists the transition from informal information to formal descriptions. As requirements discussions are conducted, stakeholders will converge on using the same set of terms to make sure that they are all talking in the same language. These unified common terms are then captured in the Term Dictionary so people can start to use formally define terms instead of unconstrained natural language. We will describe the role that the Term Dictionary plays in the transition of informal information from hyper documents to formal specification language constructs in the next section on evolution transformations.

## EXTENDED RASE CAPABILITIES

We mentioned earlier that one of the innovative aspects of RASE is the synergy it creates by integrating multiple technologies. In this section we describe RASE operations that work across tool boundaries. These operations involve work objects that are managed by different tools or require the collaboration of multiple tools. We will describe how the synergy created by the integration of multiple tools makes RASE capable of directly supporting requirements activities which was not possible with individual technologies.

### Critics

Critics can be thought as automatic agents that

analyze a design [7], inform users of the status or possible anomalies of work objects. Critics can be used to evaluate software designs against design guidelines, verify their compliance with design styles, and identify potential problems, etc.. RASE supports critics that work in the requirements domain. Currently RASE has a number of critics that analyze REMAP discussion structures.

When a critic has identified a situation that needs the user's attention, it creates a *critic resolution task* (or *CRT* for short) and post it on the user's personal agenda. A CRT is both a description of the problem that the critic discovered and a list of alternative resolutions which can resolve the problem. The user has the choice to select one of the resolutions, ignore the CRT, or fix the problem manually. Selecting a resolution results in invoking a evolution transformation which automatically or interactively corrects the problem. Ignoring the problem leaves the CRT on the user's agenda as a reminder of a problem which needs to be addressed eventually. Finally, the user may decide that none of the suggested resolutions are appropriate and make some other change to resolve the problem. In this case, the critic is responsible for monitoring the work object and removing the CRT from the user's agenda once the problem is resolved.

The following are examples of RASE critics:

*Identify Unresolved Issues*

An unresolved issue in a REMAP discussion is one for which a decision has not been made. When the RASE critic "Identify unresolved issues" finds such an issue, it creates a CRT. There are two explanations for this problem. One is that people have not started discussing the issue yet. The other is that the issue has actually already been identified and resolved elsewhere. Because of the two common explanations, there are also two possible resolutions for the problem. One is to create a decision to resolve the issue, the other is to combine the issue with another one.

We list a few more RASE critics here and briefly mention their function without going into details.

*Generate Notification*

This critic sends notifications to users when events of interest happen. For example, when a new position is created that responds to an issue, this critic sends a notification to the owner of the issue.

*Check Ownership*

This critic checks if all issues in a discussion have been assigned an owner. That all issues have owners is important to ensure that relevant discussions will be started and that the issues will be resolved.

**Evolution Transformations**

In the KBSA community, the notion of evolution transformation (ET) traditionally referred to high level editing operations that maintained the semantic integrity of a specification [3]. For instance, the operation that changes a variable name and updates all references to the variable is an ET. Another example could be one that promotes a relationship between two classes in an object-oriented design to a class by its own right, and properly updates all references to the original relationship. ET's are not necessarily semantics-preserving, since application of ET's in general evolves the specification from higher level abstraction to lower level details, resolves ambiguities, eliminates incompleteness, etc.. ET's have the advantage of being at the high level and maintaining semantic integrity, e.g., using ET's can eliminate human errors such as forgetting to update references to a renamed variable.

We have extended the notion of ET's to apply to all types of ADM first class objects, instead of just formal specification objects. In this section, we describe several interesting RASE ET's that work in the requirements domain. The RASE ET's provide very high level, usually cross-tool operations on requirements objects to transform less formal concepts to more formal ones.

The following is an example of an ET:

> *Promote Term Definition*
>
> RASE provides two ET's that work together to automate the process of identifying terms from hyper-documents, capturing rationale of why terms were nominated, inserting them into the term dictionary, and creating traceability links between the original phrases in hyper-documents, rationale information, and definitions in term dictionary.

Consider the case when an analyst examines a

requirements hyper-document, she finds that a phrase in the document defines an important concept and therefore should be formally defined in the Term Dictionary. She can highlight the phrase, say "truck manufacturer" in the hyper-document, and invoke the first ET, "Propose new term", from a user interface menu. This ET will: 1) create a new issue in the REMAP discussion that is dedicated to the nomination of terms, automatically giving the new issue an appropriate default name; 2) create a CRT on the analyst's agenda, the CRT will have one associated possible resolution. This resolution is another ET. This second ET is to create a default position and a default decision responding to the newly created issue that call for the inclusion of the term in the term dictionary. This ET will also create a new term definition entry in the term dictionary, create a hyperlink from the original phrase to the term definition, a use reference hyperlink from the term definition to the phrase in the hyper document, and another object link between the decision and the definition. The analyst may or may not choose to resolve the CRT using the suggested resolution. If she does not, a full multi-party REMAP discussion can be started on whether the term should be included in the term dictionary, starting from the issue that has already been created. However, if she does choose to use the suggested resolution, aforementioned position, decision REMAP nodes, the term definition entry, and the object links will be automatically created.

The advantage of using this critic and associated CRT is that

- it ensures that the term created has traceability links back to the rationale information in the REMAP discussion, as well as relevant information in the hyper document where the term was identified;

- the process is automated and frees the user from remembering to generate relevant traceability elements.

This example illustrates how RASE ET's and critics together accomplish high level semantic operations which can be tedious to perform manually.

We briefly describe a few more ET's here without going into details.

*Promote Term to Specification Language Concept*

There are also two RASE ET's that work together to promote a term in the Term Dictionary to a concept in the formal specification language, e.g., a class, relation, attribute or operation. These ET's work in a similar fashion as the previous two. They create specification language objects as well as related rationale objects and traceability links. These and the previous ET's are examples of how RASE supports transformation of initially informal information to descriptions of different degrees of formality.

*Create Use Reference Hyper Links*

This ET, when given a user selected term in the term dictionary, finds all uses of the term in specified hyper documents and creates hyper links from the term definition to the places where it is used.

*Export Information to a REMAP Discussion*

We mentioned earlier that hyper nodes in a hyper document can be given types such as issue or assumption. This ET exports all hyper nodes that are so marked to a REMAP discussion, automatically creating REMAP nodes of the corresponding types and creates traceability links between the hyper nodes and the REMAP nodes. This is useful because initial systems requirements are usually captured in textual format. As an analyst reviews a document, she may categorize different hyper nodes into types. The raw information in the hyper document need to be discussed before consolidated requirements are obtained and this ET prepares the inputs to the discussion.

*Term Evolution*

This ET finds all uses of an old term in specified hyper documents and changes them to the new term. This is convenient when a concept is given a new name and when there have already been documents using the old term. The use reference hyper links will be maintained after the operation.

## IMPLEMENTATION INFORMATION

RASE is being built as part of the Knowledge-Based Software Assistant (KBSA) / Advanced Development

Model (ADM) Project[1] [17]. The KBSA/ADM Project is building an integrated object-oriented design and development environment that provides automated assistance to individuals and teams spanning the entire life-cycle of large software projects. In the current phase of KBSA/ADM, we are building a field-prototype environment that is able to support a 4 to 5 person team to develop systems of about 50 KLOC in size for real life applications. The ADM environment currently consists of three components: RASE, ALE (ARGO Language Environment), and IPSE (Integrated Performance Support Environment). These components address complimentary aspects of software development. RASE supports capturing of informal information, i.e., information in the form of free format hypertext and multimedia elements, structured discussions, and terms important to the project. ALE supports the formal specification and design of systems by providing a high level object-oriented specification language ARGO with rich semantic modeling constructs. ALE provides graphical manipulation of ARGO specifications using OMT-like notation. IPSE supports project task management at both the project team level and individual developer level. At the team level, it supports project planning and decomposition by integrating and exchanging data with Microsoft Project™. At the individual level, it supports personal agenda management by informing the developer the project tasks that she is assigned to perform and allowing direct enactment of these project tasks from the agenda.

**System Information**

Currently the ADM tool set works on SUN SPARCstations™ under Solaris™ 2.4 environment. The tool set uses a commercial object-oriented database system, ObjectStore™ from Object Design Inc. to store software life-cycle artifacts, e.g., requirements and design documents, ARGO packages and specifications. ObjectStore provides persistency and versioning of stored objects. User interface of all ADM tools are built on top of the commercial graphical user interface library Galaxy/C++™ from Visix Software. The ADM tools also use PowerBroker™ for inter-process communication.

[1] The KBSA/ADM Project is sponsored by US Air Force Rome Laboratories under contract number #F30602-93-C-0015.

**The ADM Technical Architecture**

The ADM technical architecture provides common infrastructural capabilities that are reusable across all ADM tools. Three such capabilities that worth particular mentioning are *model-view framework*, *object linking* and *remote messaging*. These capabilities are key to the realization of the functional capabilities of RASE.

The model-view framework provides basic capabilities for building views (i.e., graphical user interface windows) based on the models (i.e., contents stored in persistent database). The model-view framework automatically synchronizes all views with the model when contents of the model change. All ADM Tool views are always consistently reflecting the contents of the (persistent) model. As a user invokes operations from the user interface to change design artifacts, changes are directly made to objects in the model and views are automatically updated. Therefore, there is no separate operation such as *save* in ADM to save what the user sees in views to the database.

Object-linking provides the capability for any ADM work object to refer to another ADM work object, whether the two objects are managed by the same tool or different tools. The object-linking mechanism makes it possible to create and traverse such cross-tool object links. An ADM tool user can select any two work objects via the graphical user interface windows and link them. When the user traverses an object link via the user interface, the object-linking infrastructure automatically displays the target object of the link in the original view in which the link was originally formed, invoking another ADM tool if necessary.

Remote Messaging provides a mechanism for inter-process communication. It also defines a hierarchy of standard remote message types. Object-linking uses remote messaging to create and traverse cross-tool object links.

**RELATED WORK**

As we mentioned, RASE integrated and extended a number of requirements technologies and created capabilities which were not possible with the individual technologies. These constituent technologies include hypertext (e.g., Garg and Scacchi [5]), structured discussions (e.g., gIBIS of Conklin and Begeman [10], REMAP of Ramesh and

Dhar [1], Win-Win model of Boehm et. al. [4]), design critics of Fischer et. al. [7], and evolution transformations from the KBSA community [15, 3].

Although all structured discussion approaches share similarities, they differ in their emphasis and scope. gIBIS [10] is a simple conversation model that identifies conversation elements as issues, positions and arguments. REMAP [1] extends the gIBIS approach to include the context (inputs and results) of discussion. The Win-Win model and system [4] has a great emphasis on the reconciliation of different viewpoints of multiple stakeholders. It views the requirements process as the negotiation of different stakeholders to reconcile their individual win conditions and requirements as some type of combinations of different win conditions. Depending on the nature of a project and its environmental conditions, one model may work better than others. One model that works better in one project may not be the most suitable in another.

The Nature (Novel Approaches to Theories Underlying Requirements Engineering) project applied AI techniques to requirements engineering. It focused on reconciling heterogeneous representations of requirements.

## CONCLUSIONS AND FUTURE DIRECTIONS

RASE has integrated a multiplicity of technologies to address problems related to requirements acquisition activities. These technologies include hypertext and multimedia [12, 8], structured discussion [1, 2, 4, 10], evolution transformation [3], and design critics [7]. RASE supports

- capturing of information in informal form from multimedia elements (e.g., audio or video recordings of client meetings) to natural language text;
- multi-party structured discussions following the IBIS/REMAP model;
- capturing of important terms;
- evolution transformation of informal information to formal representations;
- maintenance of traceability links among requirements objects and their rationale.

The uniqueness and contribution of RASE is not in these individual technologies, but rather, in the synergy it creates:

- RASE offers a spectrum of notations of varying formality (free style natural language, unified terminology, object models). Analysts can use the notation that is most appropriate at a particular stage of development.
- RASE defines model-spanning evolution transformations that evolve requirements from informal to formal.
- RASE maintains traceability information and evolution history of requirements objects *as* they are transformed.

This synergy allows RASE to overcome problems of using individual technologies and solve problems which was not previously possible with individual technologies.

Even though in this paper we discussed the RASE technologies in the context of requirements acquisition, they can and in fact should be used in other life-cycle stages as well. Natural language documents are used to describe initial requirements, consolidated requirements, designs, and implementation notes. As Ramesh and Dhar pointed out in [1], the REMAP model is useful for any work that involves a deliberation process. The inputs to a REMAP discussion can be initial requirements and the outputs be consolidated requirements, the inputs can be the consolidated set of requirements and the outputs be designs, or the inputs can be designs and outputs be implementations. Term Dictionary can be used to capture terms used in the design and implementation stages as well as the requirements acquisition stage. Therefore, with RASE, all life cycle work objects (requirements, designs or implementations), their rationale, project tasks that create these objects, and objects links between these objects and tasks form a web of information that is interconnected with semantically-typed links. This web of information provides a mechanism for integrating, organizing, indexing, and retrieving the enormous amount of information that is captured and generated during requirements, design, and development activities. This interconnection provides contextual and rationale information that is important to the understanding of individual pieces of work, as we pointed out in the problem statement section.

RASE has shown the potential of integration by

integrating a number of existing requirements technologies. By no means is RASE as it is now able to cover all aspects of requirements engineering. For example, although the REMAP technology help identify individual requirements, it does not provide a framework for evaluating and prioritizing them. Therefore, integration with QFD [11] and trade-off analysis (e.g., [13]) types of technology would further extend its coverage and power. Another direction for future work is to support customizable discussion models. As we pointed out previously, no single discussion model is the best under all situations. Supporting a structured discussion tool where the discussion model is parameterized and instantiated by a project would offer this flexibility.

## ACKNOWLEDGMENTS

Steve Sparks contributed many important ideas to the RASE vision. George Ding and Reuben George implemented many features of RASE. The authors would like to thank Chris Faris, the ADM project manager, for his support of this work. They also thank all other ADM team members, particularly Enaganti B. Naidu and Ilango Radhakrishnan of the Technical Architecture team, for providing the technical infrastructures that made the implementation of RASE possible. The authors also would like to thank Ming-June Lee who offered valuable inputs and help during the preparation process of this paper.

## REFERENCES

1. Ramesh, Balasubramaniam and Vasant Dhar, Supporting systems development by capturing deliberations during requirements engineering. *IEEE Trans. on Software Engineering*, Vol. 18, pp. 498-510, Jun. 1992.

2. Potts, Colin, Kenji Takahashi, and Annie I. Anton, Inquiry-Based Requirements Analysis, IEEE Software, March 1994, pp. 21-32.

3. Johnson, W.L., Feather,M., "Using Evolution Transformations to Construct Specifications", in Automating Software Design, edited by Lowry and McCartney, AAAI Press, 1991.

4. Boehm, B.W., P. Bose, E. Horowitz, and M.J. Lee. ``Software Requirements Negotiation and Renegotiation Aids: A Theory-W Based Spiral Approach." In Proceedings 17th International Conference on Software Engineering, pages 243--253. ACM Press, April 1995.

5. Garg, P. and W. Scacchi, "On Designing Intelligent Hypertext Systems for Information Management in Software Engineering," Hypertext'87, 1987.

6. Nuseibeh, B. J. Kramer, and A. Finkelstein. ``A Framework for Expressing the Relationships between Multiple Views in Requirements Specification." IEEE Transactions on Software Engineering, 20:760--773, October 1994.

7. Fischer, G. et al., ``Supporting Software Designers with Integrated Domain-Oriented Design Environment." IEEE Transactions on Software Engineering, pages 511--522, June 1992.

8. Takahashi, K. et al. "Hypermedia Support for Collaboration in Requirements Analysis". In Proceedings Second International Conference on Requirements Engineering, pages 31-40. IEEE Computer Society Press, April 1996.

9. Pohl, K. et al. ``Applying AI Techniques to Requirements Engineering: The NATURE Prototype ." In Proceedings ICSE-Workshop on Research Issues in the Intersection Between Software Engineering and Artificial Intelligence, May 1994.

10. Conklin, Jeff, and Michael L. Begeman, gIBIS: A Hypertext Tool for Exploratory Policy Discussion. ACM Transactions on Office Information Systems 6(4), pp. 303-331, October 1988.

11. Akao, Yoji, Quality Function Deployment: Integrating Customer Requirements into Product Design. Productivity Press, Cambridge, MA, 1990.

12. Christel, M., Wood, D., & Stevens, S. Applying Multimedia Technology to Requirements Engineering. In: Proceedings of the Sixth Annual Software Technology Conference. Salt Lake City, UT: Software Technology Support Center, April, 1994 (CD-ROM proceedings).

13. Boehm, Barry, Software Engineering Economics, Chapters 5-9, pp. 57-163, Prentice Hall, 1981.

14. Knuth, D.E., "Literate Programming," The Computer Journal, Vol. 27, No. 2, pp. 97-111,

1984.

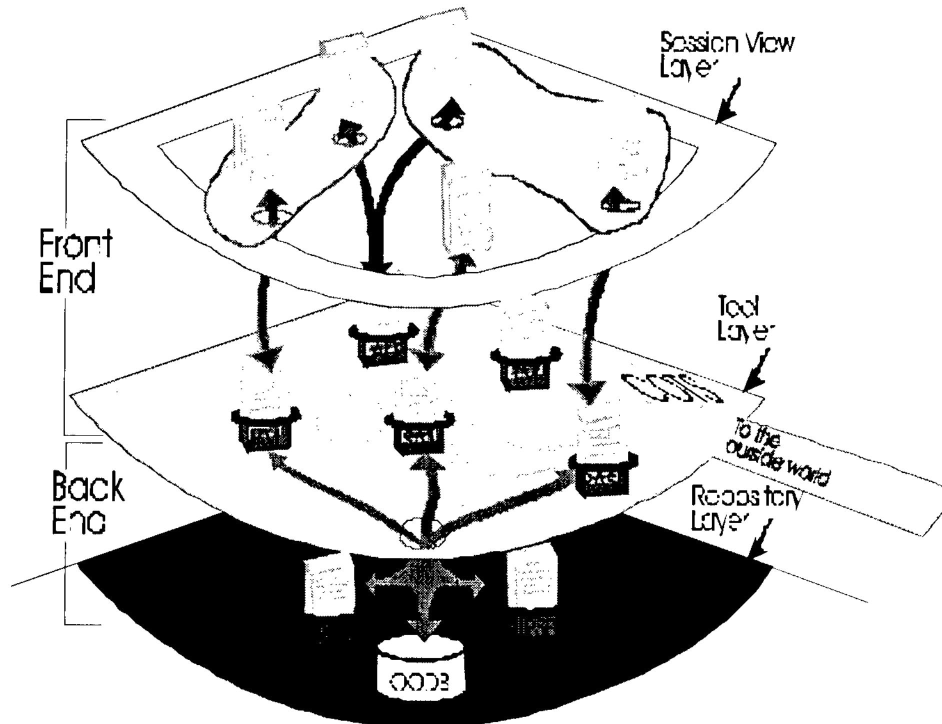
15. Green, C., D. Luckam, R. Balzer, T. Cheatham, and C. Rich, "Report on a Knowledge Based Software Assistant,", Technical Report KES.U.83.2, Kestrel Institute, June 1983.

16. Leite, Julio Cesar S P. "A Survey on Requirements Analysis", Advanced Software Engineering Project Technical Report RTP-071, University of California at Irvine, Department of Information and Computer Science, June 1987.

17. Benner, K.M., "Addressing Complexity, Coordination, and Automation in Software Development with the KBSA/ADM," In proceedings of The 11th Knowledge-Based Software Engineering Conference, Syracuse, New York, September 25-28, 1996.

# Appendix 3

## Evaluation of a Knowledge Based Software Assistant Advanced Development Model




KBSA - Architecture
(Session View Paradigan)

**Figure 12**

**James Fawcett, Ph.D., Principle Investigator,**
**Benjamin Brunk, Kiran Ganesh, Udayan Parvate**
**Department of Electrical Engineering and Computer Science**

**Syracuse University**
**31 May 1997**

**Table of Contents**


1.0 Executive Summary ---------- 54

2.0 Evaluation Strategy ---------- 56

3.0 KBSA Technology ---------- 58

3.1 KBSA Vision ---------- 59

3.2 Scope of Work ---------- 61

3.3 Past KBSA Efforts ---------- 62

3.4 References ---------- 62

4.0 Using the ADM to Assist Development ---------- 63

4.1 Summary of Goals of Proposal ---------- 63

4.2 An Introduction to the ADM ---------- 64

4.3 Functional Description of the ADM ---------- 67

5.0 ADM Logical Models ---------- 86

5.1 Software Development Model ---------- 86

5.2 Project Model ---------- 87

5.3 Communication Model ---------- 87

5.4 Collaboration Model ---------- 89

5.5 Product Model ---------- 90

5.6 Repository and Versioning Control ---------- 91

5.7 Substrate Model ---------- 93

6.0 ADM Architecture – Physical Models ---------- 96

6.1 Individual Tool Architecture ---------- 96

6.2 ADM Frameworks ---------- 97

7.0 Adding a New Tool to the ADM ---------- 103

7.1 The PART Functional Model ---------- 103

7.2 Conceptual Model for PART ---------- 105

7.3 PART Basic Types ---------- 106

**Table of Contents (Continued)**


7.4 Part Knowledge Structure ---------- 110

7.5 Problems Encountered ---------- 116

7.6 Results and Conclusions ---------- 117

8.0 RABS Maintenance Activity ---------- 119

8.1 RABS Overview ---------- 119

8.2 RABS Logical Model ---------- 119

8.3 RABS Architecture ---------- 120

8.4 RABS Design and Implementation ---------- 120

8.5 Use and Evaluation of ADM Tools ---------- 125

8.6 Results and Conclusions of RABS Activity ---------- 134

9.0 Final Conclusions ---------- 136

App-A Part Design Document ---------- 140

App-B Glossary of Terms ---------- 163

# 1.0 Executive Summary

This report documents the results of an evaluation of the Knowledge Based Software Assistant Advanced Development Model (KBSA/ADM), developed by the CSTAR branch of Andersen Consulting Company. The evaluation was conducted in the period beginning 1 January 1997 and ending 31 May 1997. Andersen funded the work as a subcontract to their prime contract with Rome Development Laboratory.

The ADM is a multi-user tool designed to support the collaborative development of software, using an incremental, iterative, and evolutionary design process. The goals of this development were to:

provide a suite of intelligent, process driven, integrated software development tools

develop a construction and integration framework in which tools may be built and integrated

The goals of our analysis were to:

understand ADM conceptual models and architecture

use the ADM in a valid context

extend the ADM by adding a new tool

provide a critical analysis of the ADM with respect to each of these activities

submit a final report (this document) detailing our findings

We want to acknowledge the vigorous support we received from Andersen personnel during the course of this study. This included training and design sessions, held in Syracuse, weekly teleconferences during the busiest part of our analysis process, extended use of equipment, and responses to salvos of our questions.

We, at Syracuse University, charted the course for this investigation, developed its methods, carried out its activities, and drew our own conclusions in an atmosphere of open inquiry.

The results of the study are compiled in summary paragraphs in Sections 7.0 and 8.0, and all of Section 9.0. Here is a synopsis of these findings:

> The ADM provides a flexible tool substrate with smoothly integrated repository and communication services.
>
> It contains an extensible tool layer which provides software assistant functionality. Individual tools are part of a federation with all the support services necessary to inter-communicate and cooperate to achieve a common goal. All are governed by a session layer supporting project management and user view consistency.
>
> Use of a session manager to implement the model-view-controller paradigm and coordination through integrated process support to manage task enactment is a major success for the ADM. It supports the process model advocated by the ADM without intruding on the users' focus on their activities.
>
> The ADM has a pleasing user interface, smoothly integrating its tools into its support model.
>
> Some of the individual tools are weak, not achieving their software assistant objects very well. In one case technology used in the ADM has been superceded by commercially available tools.
>
> The ADM is an advanced development model. It could not be used effectively in a production environment without significant polishing and substitution of some of its tools for better-developed functionality.
>
> It has provided a proof of concept for the management of collaborative work and capturing of requirements, assumptions, and design decisions in an iterative development environment.
>
> We especially like the concept of object linking and the use of an object oriented database to capture complex relationships between the large number of products in even a moderately sized development project. This could be enormously useful for management and tracking of an evolving product baseline in complex software development projects.

## 2.0 Evaluation Strategy

In our evaluation proposal we said:

"Our intent is to test, evaluate, and report on the effectiveness of the KBSA paradigm and implementation to support software development and promote productivity and product quality. We will focus on the robustness, effectiveness, and applicability of the ADM implementation to support several types of software development. We will also evaluate the potential of its concept and architecture to provide a framework for building tools supporting all phases of the software development process."

The goals of the evaluation and strategy we adopted to accomplish them are:

**Understand the ADM conceptual models and architecture:**

We devoted the first six weeks of this six month study to learn the Advanced Development Model paradigm, architecture, and as much of the implementation as practical for this brief analysis. For this activity we incorporated two Solaris workstations and an NT machine into the Syracuse University network system, configured to support ADM processing. Andersen personnel conducted an intensive 3 day training session and provided us with documentation for the ADM and its tools. During this period we explored the ADM tools, trying to understand the goal and functional capabilities of each.

**Use the ADM in a valid context:**

To achieve this goal we selected a software maintenance activity conducted on a modestly complex software product most of the team was familiar with. The product is a Repository and Build Manager System (RABS) developed in CSE784 - Software Studio class in the Fall of 1996. Two of the team members had participated in its development. We knew this software system was operational, well organized and documented. We identified nine latent errors in its implementation that needed to be eliminated for the product to be generally useful. In addition there were some modest changes in functionality we wanted to implement.

Our team fit the ADM paradigm, e.g., four developers, each skilled in the process and implementation of software, and working in a collaborative environment toward common goals. Our strategy was to follow the ADM development process, using each of the ADM tools according to its design intent to conduct this maintenance activity. The results of this phase are

documented in Section 8. of this report.

**Extend the ADM by adding a new tool:**

One important issue concerning the ADM implementation is just how difficult is it to add a new tool to the ADM tool federation? We decided to attempt to add a new tool to the ADM which extends the ADM's model of the software development process. This we felt would test not only the ADM implementation but also test the flexibility of its architecture and conceptual models.

We call the new tool Project Archival and Report Tool (PART). PART introduces a new knowledge structure intended to capture and manage each of the products of a developing software baseline, e.g., requirements, design, and test documentation, and code for the developing product as well as test and qualification drivers. The archive provides the structure to manage product components, information about each of the components, and critical analysis and problem resolution information. The tool is intended to support build processes for both software and documentation from component pieces which closely reflect the organizational structure of the development team and software architecture.

**Provide critical analysis of the ADM with respect to each of its activities:**

Our analysis divided the ADM into its layered structure, partitioned along the lines of its major tools and frameworks. For each of these we developed, with the help of the Andersen team, a model of the tool or framework based on the activities it is intended to support. We then attempted to exercise that element the way it was supposed to be used. We recorded our observations and compared the results with both the component's model and our own perception of its utility in a software development process.

These observations and conclusions are recorded in Section 7. for PART addition, in Section 8. for the RABS maintenance activity, and in Section 9. for overall ADM functionality.

**Submit a final report detailing our findings:**

This report is the culmination of the evaluation effort.

## 3. KBSA Technology

The KBSA program is an effort to provide automated assistance to individuals and teams of software developers spanning the entire life cycle of large software projects. A variety of research efforts have made significant progress towards realizing and refining the vision for this program since the first time it was outlined in [GR83].

Knowledge Based Software Engineering is another name for applying AI to software problems, emphasizing the fact that creating software is a knowledge-intensive activity. There are four primary reasons why software engineering is an interesting area for AI research [SE92]. They are:

1. Writing large software systems is a complex activity that requires a great deal of individual and organizational intelligence.
2. An explosion of tools and techniques, such as CASE tools, object-oriented programming methodologies, fourth generation languages, visual languages, and a variety of new software development environments have been devised to support development of large complex software systems. Yet managing the development of very large systems remains a difficult and sometimes treacherous endeavor.
3. Software problems combine issues that have been studied in isolation, such as human interface problems, computer supported cooperative work, basic AI representation issues, knowledge retrieval, reuse problems and visualization

It is the amount and scope of relevant knowledge required for development that makes the implementation of software so difficult. Creating a large software system requires a knowledge of the domain, the final implementation platform (of both hardware and software), the current software process, the details of all interdependent components, and personnel resources. The KBSA approach assumes that making more knowledge available to individual programmers, teams and managers will accelerate the timely production of high-quality software.

### 3.1 KBSA Vision, according to Rome Laboratory

According to the KBSA vision outlined in [GR83], the following requirements must be satisfied by an ideal KBSA tool:

- All software life-cycle activities must be machine mediated and supported
- Development assistance must formalize this process which will enable maintenance to be performed by altering the specification and replaying the previous development process.
- Specification validation cycles must get the specification correct and to get the end-users to completely state their requirements before implementation is produced.
- Project Management must be supported
- Product management must be supported: Assistance to be provided for generating Requirements, Performance, Testing, Help Documentation etc.

The KBSA tool architecture must consist of different frameworks, with an activity coordinator that controls various knowledge-base managers. Related life-cycle activities must be grouped and coordinated. Agents and messaging to be used here if helpful.

The different facets of KBSA tool are

1. **Requirements**
    - Need for a formal requirements language
    - Requirements editor to create/modify requirements definitions
    - Requirements testing

2. **Specification validation**

- Specification needs to be executable
- Possible methods of testing for correctness of a specification are prototyping, static validation, dynamic validation
- Specification paraphrasing

3. **Development**
    - Need for a wide spectrum language to encompass design of a system in all stages from formal specification through optimized implementation
    - Interactive mechanical development

4. **Performance**

   - Data structure analysis and advice
   - Subroutine and Module decomposition advice

5. **Testing**

   - Test Case Maintenance Assistant: accept changes in test data, to schedule the running of relevant tests automatically when units undergo change.
   - Testing activity to be distributed over validation and development activities and will cease to exist as a separate phase.

6. **Reusability/Functional Compatibility/Portability**

   - A portability assistant to match components interfaces (imports and exports) which will point out a set of constraints to which a running program on an installation must conform and will walk-through them accepting responses on the way.

Other supporting technology areas are identified as

7. Wide spectrum program design languages (PDL)
8. Extended formal semantics
9. General inference systems
10. Domain specific inferential systems
11. Specialized inferential systems
12. Integration technology
13. Databases

    - Administrative: Agents and their relationships
    - Software: A set of modules known for a particular instance of KBSA development environment
    - Knowledge Base: Acquired by and available to various facets of KBSA

14. Toolsets

    - Editors, Compilers, Program Transformation aids, Debuggers, Tools for analysis, query, project management, message handling, database management etc.

15. User Interfaces
16. Activities Coordination

## 3.2 Scope of Work

Even though a fully integrated environment (as described above) is the ultimate goal, any current KBSA research project must carefully address some selected parts of this process. A KBSA effort must answer the following crucial questions [SE92]:

- What part of the software process is targeted?
- What knowledge is applicable, and how can it be represented, acquired, and maintained?
- How can we present this knowledge to developers, teams and managers to improve quality, cost and timeliness of software development?

Producing a large software system is a complex, multi-step process, and it results in a wealth of artifacts such as

- User requirements
- System specifications
- Code generation
- Testing Scenarios
- Documentation

Once the relevant knowledge is represented in the system, maintaining that knowledge base is as critical as maintaining the code base or document base itself. The presentation and integration of knowledge based approach into the everyday working world of software engineers is a critical challenge for the KBSA community [SE92].

### 3.3 Past KBSA Efforts (in chronological order)

The following are some of the outcomes from several past efforts by various research groups:

1. Project Management Assistant, by Kestrel Institute (1984-1986)
2. Knowledge Based Requirements Assistant (KBRA), by Sanders Associates (1985)
3. Knowledge Based Specification Assistant (KBSA), by University of Southern California Information Science Institute (USC ISI)
4. ARIES system, by USC ISI with Lockheed Sanders (1988)
5. Performance Optimization Assistant, by Kestrel Institute (1988)
6. KBSA framework with combination of Common Lisp Object system/Logilisp, KBSA User Interface Environment (KUIE), Configuration and Change Management Model (CMM), by Honeywell System Research Center (1986)
7. Development Assistant based on KIDS, by Kestrel Institute (1988)
8. Transaction Graphs and Artifact Configuration Management System for unified formalism in coordinating and managing products & processes using the KBSA paradigm, by Software Options (1988)
9. KBSA concept demo, by Andersen Consulting (1988-1992)
10. KBSA/ADM initial operational capability by integrating previous efforts of KBSA technologies to form a working environment, by Andersen Consulting (1992)

### 3.4 References

[GR83] C. Green, D. Luckham, R. Balzer, T. Cheatham and C. Rich, Report on a Knowledge Based Software Assistant, RADC-TR-83-195, August 1983

[SE92] Peter G. Selfridge, Knowledge Based Software Engineering, IEEE Expert, December 1992

In addition, proceedings from the following conferences have provided valuable information:

- SEKE: Software Engineering & Knowledge Engineering Conference
- KBSE: Knowledge Based Software Engineering Conference
- AAAI Workshops on Automating Software Design
- Workshop on Applying AI to Software Problems

# 4. Using the ADM to Assist Development

## 4.1. Summary of Goals of the Rome Laboratory Proposal

The goal of the KBSA/ADM effort is to develop an operational Model that has the ability to fuse concepts and techniques of prior efforts to enable use and evaluation of the KBSA's new life-cycle methodology.

### 4.1.1. Technical requirements

The requirements are grouped into the following:

- Functional/non-functional requirements
- Software Management Requirements
- Demonstration and Assessment requirements

### 4.1.2. Functional/non-functional requirements

The KBSA/ADM should provide intelligent assistance and automation throughout the software life cycle. Major concerns are completeness, quality and maturity of technology and products to be used or produced and the overall usability of the ADM by prospective KBSA users.

Functional requirements include providing total software life cycle support from the initial delineation of user requirements through post deployment support. It should also provide machine mediation, communication, coordination, monitoring, analysis, documentation, and automated assistance for the activities of requirements acquisition and design creation, discovery, modification, explanation, and implementation.

### 4.1.3. Non functional requirements

- Support for simultaneous and interactive use by a development team consisting of 1-4 people.
- Informative and user-friendly interaction
- Support for application developments of 50,000 or more lines of code.
- The use of prevailing standards
- Extensible and modular design of the ADM as an open system, enabling it to continue to evolve following delivery and to integrate with other software development tools

### 4.1.4. Software Management Requirements

A process should be established for managing the design, development, and review of the ADM. Most important are the use of the management methods and capabilities that minimize the risk and maximize project visibility for both contractor and government managers.

### 4.1.5. Demonstration and assessment requirements

Demonstration of the ADM by example application development shall be sufficient to exercise and illustrate all facets of the KBSA ADM life cycle, including the reuse of design knowledge and impact on the post deployment support. Test and evaluation shall be conducted to assess the reliability, functionality capacity and limitations of the ADM and the KBSA process. The tests should collect sufficient data to allow extrapolations and projections regarding the extensibility of the KBSA technology. Testing should be based on experimental design methods to assure statistically adequate and sufficient coverage of the evaluation domain

## 4.2. An Introduction to Andersen Consulting's ADM

### 4.2.1 Introduction

The previous section described the Rome Laboratory's vision for providing automated assistance to individuals and teams of software developers spanning the entire life cycle of large software projects. This section describes the specific objectives of the Advanced Development Model (ADM) developed by Andersen Consulting towards the fulfillment of these goals.

#### Issues that the ADM claims to address

There are various problems associated with building and evolving large scale software systems. Some of the problems fall under the headings of complexity, collaboration and automation. The problems under each of these categories can be briefly summarized as follows:

## Managing Complexity

The complexity of a large software project arises from four critical elements: (1) The complexity of the problem domain (2) the difficulty of managing the development process (3) The flexibility possible through software and (4) The problems of characterizing the behavior of discrete systems. Because of this, only a small number of developers understand the entire system. This understanding is very critical for finding and resolving inconsistent requirements, developing a coherent design, factoring the total system into manageable pieces for development and evolving the system when requirements change in unanticipated ways. Critical design decisions, assumptions and rationale are seldom documented and indexed.

## Automation

In software development, inevitably there are a large number of routine tasks, which are well understood, but tedious to perform (which could result in additional avoidable errors due to their routine nature). In a small project one lives with these inconveniences, but in a large project the costs for performing these small tasks and fixing the resulting avoidable errors compound quickly. Automating routine tasks can reduce the number of errors made and in doing so, shorten development time.

## Coordination

As teams get larger, coordination becomes more essential, but harder to achieve. No matter how well a system is structured, communication among teams and within teams is necessary. The difficulty in these communications is disseminating knowledge in a timely and understandable manner to interested parties. A big part of enabling coordination is establishing for developers the proper context in which to understand the work products so that they can use and evolve them properly. The following table summarizes how the ADM proposes to address each of the three issues. The following section describes these mechanisms in greater detail.

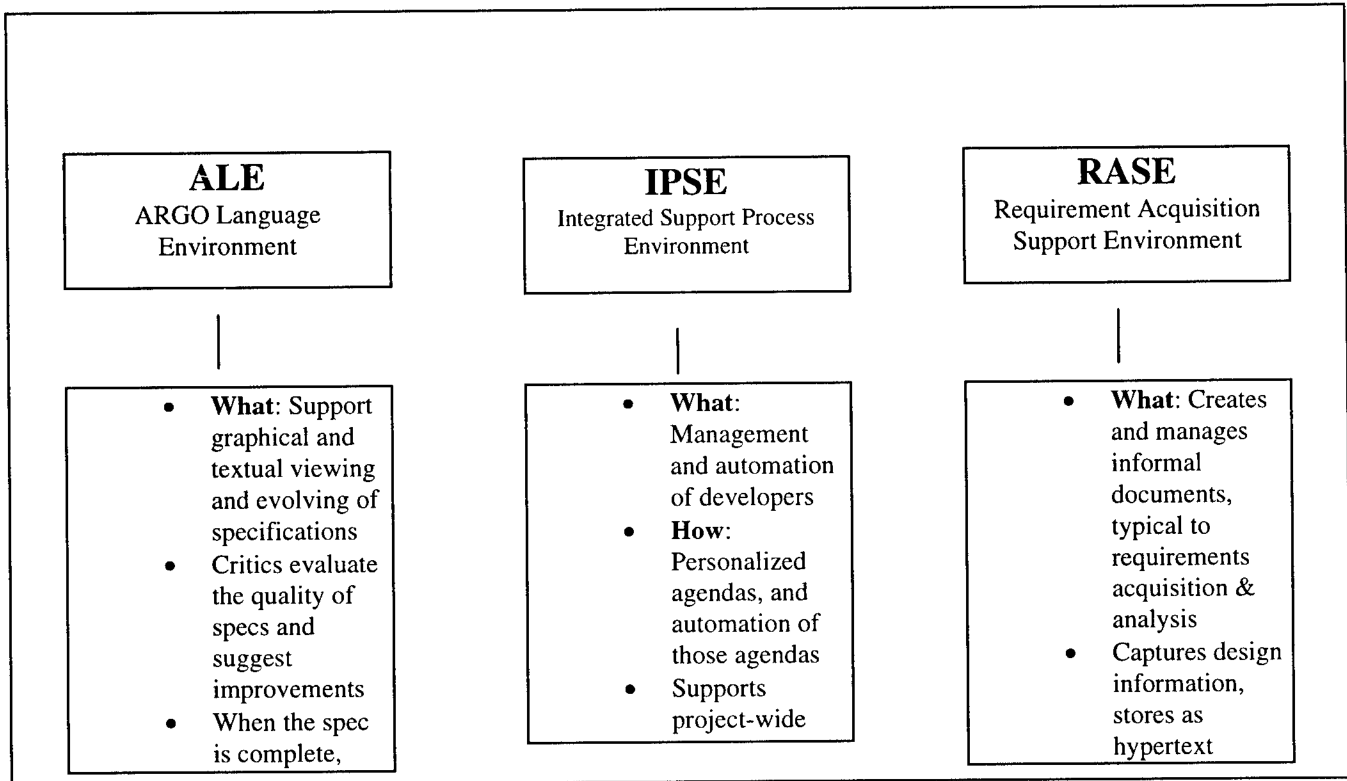
| **Manage Complexity** | **Automate What's Understood** | **Support Coordination** |
| --- | --- | --- |
| 1. Support for Appropriate Notations<br>2. Mechanisms to Maintain consistency between interdependent work products<br>3. Mechanisms to Capture Contextual Knowledge about why work products are the way they are<br>• Design History<br>• Design Discussions<br>• Hyperlinks<br>4. Explicit support for Project Task Decomposition and Enactment | 1. Evolution Transformations<br>2. Process Enactment<br>3. Critics<br>4. Code Generation | 1. Repository Management<br>2. Task Management<br>3. Structured Multi-party Discussions |

TABLE 1 - How ADM Proposes to Address Each of the Three Issues

## 4.3. A Functional Description of the ADM

This effort as two driving goals: (1) to provide a suite of integrated software development tools (2) to provide a tool construction framework in which tools may be built and integrated.

The ADM is made up of three principal environments, as shown in the figure below:




Structure of the KBSA/ADM tool

The following sections describe the use of these three tools in greater detail.

### 4.3.1. ALE

ARGO language environment (ALE) supports graphical and textual viewing and evolving of object oriented specifications. When a specification is complete ALE can generate C++ code for a system, subsystem, or component.

#### 4.3.1.1 Packages and Specifications

In the operational prototype ALE is a tool which introduces two new topic types:

Package Topic

Specification Topic

A specification is a container for packages. Simple specifications show which packages are in the current specification. The only relationship between packages is the import relationship.

A package is a container for classes. Packages show how related classes are grouped together, and allow for one level of grouping related classes together.

The following are the concepts from Object Oriented design which are supported while defining a graphical representation with an OMT diagram:

- An abstract class is a high-level method for grouping related classes together. Although the abstract class may contain default attributes and methods from which other classes are derived, instances of this class do not actually exist.
- An attribute describes a particular characteristic of an object.
- A class is a group of objects with similar properties and attributes, and common behavior, relationships to other objects, and semantics. Use a class diagram to show the relationships between classes.
- Classes can inherit attributes and methods from other classes based on a relationship. There are two commonly used types of inheritance: public and private. Public inheritance means that other packages can access the class methods and attributes of the package. Private inheritance means that only the current package can access the class methods and attributes of the package.
- In the ALE diagrammer, you can draw lines between the classes and packages that show inheritance.
- A package is a conceptual grouping of classes. The package provides a level of abstraction by grouping classes together. A package diagram view shows a specification topic. A class diagram view shows a package topic.
- A relationship shows that two or more objects have common characteristics. This is a parent-child relationship. In KBSA, the relation is a pointer kept in one object that points to the other object. In ALE, you can also designate a composition relationship. A composition relationship shows aggregation of objects. For example, when an object is composed of several objects of other types, the combined parts represent the composite object.
- A specification is the top abstraction level that describes a design model and its packages. Use a package diagram view to show a specification.

#### 4.3.1.2 Critics

The concept of critics enables the tool to evaluate the quality of the specification and suggest ways to improve it. ADM code generation translates an ARGO specification into C++ code.

There is a knowledge based assistant function that automatically generates tasks for you when something is missing from your class structure. For example, if you delete a class and do not delete a corresponding relationship, the ALE critic sends a message to the KBSA Session Manager. The Session Manager then adds a resolution to your plan that requires you to either delete the dangling relationship or add the class back into the design. Currently there are three critics:

- Content critic
- task Completion critic
- Cohesion and Coupling critic

The content critic evaluates the package for its correctness. In addition, it generates some possible resolutions and informs the user through IPSE resolution mechanisms. Currently there is no framework support for developing critics in the KBSA/ADM environment.

#### 4.3.1.3 Tool Usage[1]

1. Define Packages and their relationships

To create a package diagram in the Session Manager window

- From the Topic menu, select Create.
- From the cascading menu, select Specification.
- Type a name for the topic in the Topic Name field and click OK.
- From the View menu, select Create.
- From the cascading menu, select Package Diagram.
- Type a name for your view in the View Name field.

To create a package

- Click the package tool in the tool palette (white square).
- Pressing and holding the left mouse button, drag the mouse down and to the right. Release the mouse button.
- In the Package Name field of the Package Editor window, type a package name.

[1] This material is extracted from KBSA help files.

- Click OK.

To delete a package

- Select the package you want to delete.
- From the Edit menu, select Delete.

To modify a package

- Select the package you want to modify.
- From the Edit menu, select Update.... The Package Editor window appears.
- Type a new package name or edit the structured comment.
- Click OK.

To view the contents of an imported package

- Select the package.
- From the Specification menu, select Package Textview. The ARGO syntax for the selected packages and any current link errors appear.

To show an inheritance relationship between packages

- Click the inheritance tool in the tool palette (black triangle is private inheritance and other is public inheritance).
- Press and hold the left mouse button over the source package.
- Drag the mouse over the target package, then release the button. A line showing the inheritance relationship is formed between the two packages.

## 1. Defining Classes and their Relationships

To create or open a class diagram

- Double-click on a package icon. The Class Diagram Editor opens.

To create a class

- Select the class tool in the tool palette (solid white square is regular class and white square with a dotted line is abstract class).
- Pressing and holding the left mouse button, drag the mouse down and to the right. Release the mouse button.
- In the Class Name field of the Class Editor window, type a class name. You can also designate whether you want the class to be public, private, or protected.

- Click OK.

To delete a class

- Select the class you want to delete.
- From the Edit menu, select Delete.

To modify a class

- Double-click the class you want to modify. The Class Editor window appears.
- Change settings as required.
- Click OK.

## 3 Define Class relationships and Inheritance

To draw an inheritance relationship

- Click the inheritance tool in the tool palette (shaded triangle with a line).
- Press and hold the left mouse button over the first class.
- Drag the mouse over the second class, then release the mouse button. A line showing the inheritance relationship appears between the two classes.

To define a relationship

- Click one of the relationship tools in the tool palette (rectangle is regular relationship and other is composite relationship).
- Press and hold the left mouse button over the first package.
- Drag the mouse over the second package and release the mouse button. The Relationship Editor window appears.
- Specify the relationship name.
- Click OK. A line showing the inheritance relationship appears between the two packages.

To delete an inheritance relationship

- Select the inheritance you want to delete.
- From the Edit menu, select Delete.

To delete a relationship

- Select the relationship you want to delete.
- From the Edit menu, select Delete.

## 3 Add attribute objects to a class

To create an attribute for a class

- Double-click on the class. The Class Editor window appears.
- Click an attribute object button at the bottom of the window to open the appropriate dialog.

To delete an attribute

- Double-click the class. The Class Editor window appears. Attribute objects are listed on the right.
- Select the attribute you want to delete.
- Press Delete.

To modify an attribute

- Double-click the class. The Class Editor window appears. Attributes are listed on the right.
- Double-click the attribute you want to modify. The Modify Attribute window appears.
- Change settings as required.
- Click OK.

## 3 Link objects in the KBSA repository

To display the object link dialog

- Position the mouse pointer over a class or package. Press and hold the right mouse button.
- From the popup menu, select View Object Links. The Object Link window displays the links for the class or package you selected.

To define an object link

- Select the class or package from which you are linking.
- From the ObjectLink menu, select SetAsSource.
- Open the view that contains the object you want to link to the class or package.
- Select the object you want to link. For example, if you want to link Hypertext in the Hyper Document Editor, highlight the text you want to link.
- From the ObjectLink menu in the current view, select SetAsTarget.
- From the ObjectLink menu in either view, select FormLink.

3 Generate Code

To generate C++ code

- In the Package Diagrammer, select the Specification menu.
- From the Specification menu, select Generate. The Save As window appears.
- In the Save as: field, type your C++ file name, giving it either a .cpp (source code) or .hpp (header/declarations) extension.
- Click OK. Your files are now saved in the directory you specified.

3 import predefined concepts

To import a package

- From the Specification menu, select Import....
- Select a package from the repository. The imported package appears on your package diagram.

#### 4.3.2. IPSE

IPSE, the Integrated Process Support Environment, provides the collaborative underpinnings for the ADM environment. Its fundamental goal is to provide process driven support for a software development team. IPSE's meta model includes the concepts of task, plan, resource, deliverable, dependency, and resolution. It addresses the problems of planning and executing complex interdependent activities (iterative and incremental) and provides the infrastructure to allow collaboration across groups of developers to take place. The collaboration model takes the form of managing work objects amongst developers, and providing communication and coordination between team members across time and

space. In addition, IPSE also provides a means of preserving corporate knowledge by updating development histories for work objects, updating dependencies between objects, and by supplying ongoing project assessment.

### 4.3.2.1 IPSE Interaction with MS Project

IPSE utilizes the commercial off-the-shelf (COTS) tool Microsoft Project® (MSP) in order to provide traditional project management views and for creation of the project work plan. A project is a structured relationship of tasks assigned to resources. Resources complete tasks by creating deliverables, referred to in KBSA as topics. The Microsoft Project work plan is exported through the MSP-KBSA interface into IPSE, which decomposes the work plan information into tasks, resolutions and deliverables and then assigns resources in the ADM to the tasks. Each task is assigned to a single user (resource) in the work plan. Each task has a distinct resolution associated with it as well as one or more deliverables. A task may have an input consisting of a previous task. This imposes an execution order on the two tasks (e.g. a task may not be enacted unless its input task is completed), or it may have a deliverable as its input (in which case the deliverable must exist in order to enact the task).

Tasks, resolutions, and deliverables are linked together with Microsoft Project. Each task in the work plan has a single proposed resolution. This resolution indicates the topics the user needs to create to complete the task. In Microsoft Project, the manager can add a resolution to provide the resource with information about the task and its deliverables. Thus a resolution is some information which guides the user towards the completion of a task. Completing a task involves creating a topic (deliverable) for it, consisting of either a specification, discussion or hyperdocument. A deliverable is an object the task resource completes during a task. A deliverable can serve the following functions in relationship to a task:

- Input--Deliverables that provide information for completing a task.
- Update--Deliverables created earlier in the project that are modified by a task.
- Output--Deliverables that are produced during the completion of a task.

If you add, delete, or modify tasks or deliverables in a work plan, input and output relationships are preserved. However, the relationship between deliverables and the tasks that update them must be manually changed. Updating the index number of the task in the deliverable information does this. Additional deliverables may be added to a task from within the ADM environment.

Linkages between tasks and deliverables are what define the execution order of the tasks in the project

plan. The project work plan also defines time limits for tasks and this information is maintained in the task information structure within IPSE. If a task is past due, or if another user wishes to enact a task whose input is not yet in progress or completed, the ADM's automatic tracking mechanism notifies the "offending" user via SAM to enact the task that is blocking the flow of the project.

Microsoft Project provides a view of the project plan in the form of a spreadsheet type view as well as a graphical Gantt chart view. The project manager can use these views to get a clear idea of the current status of a project. Microsoft project cannot inform a manager that a user is unable to enact a task of theirs because it is dependent upon another task or deliverable as its input that has not yet been completed. This is because there is no coupling between IPSE and Microsoft Project after the work plan is exported via the MSP-KBSA interface. However, Microsoft Project does give a very clear overview of the entire project and the task dependencies are clearly visible in the Gantt chart view.

Another thing that the Microsoft Project work plan provides is methodological support. The methodology provided with KBSA provides an organized, repeatable process for building information systems. The methodology contains eight major phases of project development, called task packages. Each task package contains one or more tasks. A task is an activity that must be completed in sequence or conjunction with other tasks in the package. Each task may result in one or more deliverables.

The methodology is part of the Microsoft Project file ACODMESR.MPT. It forms the basis of project work plans developed for use with KBSA. ACODMESR is the Andersen Consulting base methodology, but any methodology can be invented and utilized. IPSE does not use any of the methodology information, it is merely stored within its structure.

#### 4.3.2.2 The MSP-KBSA Interface

The MSP-KBSA Interface manages the relationship between KBSA's IPSE and Microsoft Project. It provides the following options:

- Start Project - opens a new project in Microsoft Project using the KBSA template.
- Open Project - connects you to the domain name service/communication manager on UNIX and lists the sessions currently running from which you can select a target for your export or import.
- Close Project - closes the current project.
- Put Project - builds project information into a list message and exports it to the selected session.
- Get Project - retrieves a project from an active session and imports it into Microsoft Project.
- Exit - closes Microsoft Project.
- Help - opens the KBSA Project Management help.

- Methodology - opens the Methodology help.

#### 4.3.2.3 Project Management in the KBSA SAM[2]

Once you have exported a project into the KBSA environment, you can use KBSA to alter the project plan in certain ways without importing the project back into Microsoft Project. Specifically, you can add or remove deliverables to or from tasks. You can also add deliverables to the project that are not directly linked to tasks. Note that only deliverables associated with tasks in the work plan can be imported back into Microsoft Project.

In addition, keep in mind that the export process merges resource information, rather than overwriting it. KBSA retains resource assignments from the original project export, regardless of whether these resource assignments have been changed in Microsoft Project. For example, you could import a project from KBSA into Microsoft Project and change the resource for a task from Paul to Jane. When you export the project back to KBSA, both Paul and Jane are resources for the task.

#### 4.3.2.4 Creating a work plan

1. On the MSP-KBSA Interface, click Start MSP. Microsoft Project opens and creates a new project using the KBSA template.
2. Open the project in which you store your base methodology. The default file name is ACODMESR.MPT.
3. Save the project using a new name. Use the extension .MPP.
4. Delete tasks, deliverables and resolutions that are not relevant to your project.
5. Add any tasks, deliverables and resolutions that are needed for your project.
6. Rename tasks and deliverables as desired for your project.
7. Save the project.
8. Export the work plan to KBSA/ADM.

[2] This material was also extracted from the KBSA help files.

## 4.3.3. RASE

RASE creates and manages informal documents common to requirements acquisition and analysis. It also provides a semi-structured representation based on nodes and links, in which one can capture a multi-party discussion. It provides a platform to represent both loosely structured documents as well as structured discussions. The deliverable topic types supported by RASE are:

### 4.3.3.1 HYPERDOCS

A hyper-document (hyperdoc) is a KBSA document type that describes some aspect of a project and may have links to other KBSA objects. A link is a way to connect two pieces of information. The Hyperdoc Editor is a KBSA tool for creating these hyperdocs.

Hyperdocs organize information using common text processing. They also provide the ability to link to other design deliverables in the repository. This allows the user to create lattices of documents and deliverables that can be browsed in a meaningful way. With the Hyperdoc Editor, the user can capture information that is not easy to represent in a parametric format. So in addition to capturing the structured design information with KBSA tools such as ALE, the user can use the Hyperdoc Editor to manage and manipulate unstructured information.

#### *Hyperdoc Styles*

Styles are predefined standards for displaying text. One can use style to both visually format and organize information. When applying styles to text in a document, the information can be organized into structured nodes. Nodes are composed of a node title (denoted by a style selection), and a body.

#### *View Modes*

When one first creates a hyperdoc, one must select whether he/she wants to create a normal hyper document view or an outline hyper document view. A *Normal View* is where one types the bulk of the information. The *Outline View* is similar to outline view in Microsoft Word. In outline view, one can arrange or rearrange the structure of a document. Only the node name can be edited in the outline view.

#### *Display Modes*

There are two display modes in the Hyper Document Editor, namely the *Edit Mode*, and *Browse Mode*. The Edit Mode is where you can type text, change styles, create object links, form bookmarks, and export

to discussion nodes. The Browse Mode is for navigating a hypertext document by following hyperlinks. Hyperlinks can be used only in Browse Mode.

Tool Usage[3]

1. To create a document

- From the Topic menu in the KBSA Session Manager window, select Create.
- From the Create list, select Hyper Document.
- Type a name for your topic in the Create Topic window and press OK.
- From the View menu in the KBSA window, select Create.
- From the Create list, select Hyper Document Normal View or Hyper Document Outline.
- Type a name for the new hyperdoc in the Create View window and press OK.

2. To open a document

- In the KBSA Session Manager window, select the topic in which you want to create a document.
- From the View menu, select Create.
- From the Create list, select Hyper Document Normal View or Hyper Document Outline.
- Type the document name in the Create View window and press OK.

3. To Save a Document:

Click the Save button

4. To Change Text Style:

- Place the cursor on a line of text or a blank line.
  - Select the style you want to apply in the Style list.

  You can choose body or heading numbers that are less than or equal to the immediately preceding heading number, or the heading number that is greater than the immediately preceding heading number by one.

5. To Change the Text Font:

- Select the text you want to change.
- From the Format menu, select Font.
- Select the font and size in the Font Chooser window.

[3] ibid

### *Organizing a Document*

To view a document in a different display mode: From the Display menu, select either Edit Mode or Browse Mode. To create a bookmark: Select the text you want to mark, and then from the Bookmark menu, select Create Bookmark. The bookmark text turns green. To see a list of bookmarks, select Bookmark List from the Bookmark menu. Select an item in the list to go to the bookmark.

### *Object Links*

To create a hyperlink

- Select the text you want to link.
- From the ObjectLink menu, select SetSource.
- Open the view that contains the object you want to link.
- From the ObjectLink menu, select SetTarget.
- From the ObjectLink menu (in either view), select FormLink.

When the link is formed, the text in the HyperDocument Editor turns blue. To use object linking shortcuts, click and hold the right mouse button, then select the operation you want to perform. To view the link list, press and hold the right mouse button in a HyperDocument Editor view. The link list appears. To create a node and link in the Discussion Editor:

- Select the text you want to link.
- From the Operations menu, select the kind of node you want to create in the Discussion Editor.
- In the Discussion List window, select the discussion you want to link to (or type a new name and click the Create button).
- Click the Continue button. The discussion opens, and your new node is in it. The new node is already linked to the hyperdocument. You can perform this operation only if the document is in Edit mode.

To display the Object Linking Shortcut menu

- Press and hold the right mouse button.
- Select the object linking operation you want to perform.

To display the shortcut menu, select the dotted line at the top of the operation list. This places the shortcut menu on your window until you close it.

**4.3.3.2.**

## REMAP Discussions

A discussion is a way for stakeholders on a project to resolve complex design issues. System designers and customers can contribute their expertise and assumptions to a discussion to help find answers to these design issues. The KBSA Discussion Editor is a tool used to help document and illustrate discussions that take place. It helps the development team track assumptions and understand why certain decisions were made throughout the project.

A discussion in the Discussion Editor can contain the following nodes, or pieces, of a conversation:

1. **Argument:** An argument is one of the nodes in a discussion. Each issue's position can have one or more arguments. An argument can support or contradict a position. Arguments are based on or depend on assumptions.
2. **Assumption:** An assumption node is information that a stakeholder believes to be true about an issue. Assumptions are what arguments are based on or depend on.
3. **Decision:** A decision node is the result of the discussion and resolves an issue in a discussion. A Decision Node in the KBSA can record such information as how to resolve an issue, standards, or quality checks.
4. **Issue:** Issue discussion nodes represent key issues in the design problem. They are considerations that you need to address when you design your system. Each issue can have many positions. Each issue is the root of the tree or hierarchy. In a hierarchy, the children of an issue are positions, and the children of positions are arguments.
5. **Position:** A position node is a proposed solution to an issue. A discussion can contain multiple positions, and one position can respond to an issue or set of issues. Each position can have one or more arguments, which either support the position or object to it.

The Discussion Editor also shows relationships between these discussion nodes. Relationships show how the nodes are connected to each other. For example, a relationship would show which issue a given position responds to. Relationships are illustrated in the Discussion Editor with colored lines connecting the nodes. This illustration shows how parts of a discussion are related and which factors contribute to the final decisions. Each different colored line represents a different kind of relationship. You can display the legend to remind you of the meaning of the colors at any time. When you select the pointer button (or select button), you can then click on a link and its type is displayed.

**Tool Usage**[4]

**To create a new discussion**

- From the Topic menu in the KBSA window, select Create. A cascading list appears.
- From the cascading list, select Discussion.
- Type a name for your topic in the Create Topic window and press OK.
- From the View menu in the KBSA window, click Create. A cascading list appears.
- From the cascading list, select Discussion View.
- Type a name for the new view in the Create View window & press OK.
- If your discussion topic already exists, select the topic and skip 1-3.

To change the size of the view: From the Display menu, select Zoom In or Zoom Out repeatedly until the discussion is the desired size.

**To delete a discussion view**

- In the main KBSA window, select the discussion view you want to delete.
- From the View menu, select Delete. This procedure deletes only the view and not the topic.

**To change the view layout**

To automatically position the nodes: From the Layout menu, select Arrange Layout. The nodes are automatically arranged, depending on the orientation and layout type selected. To turn off automatic node placement: From the Layout menu, select Disable Layout. You can now place the nodes anywhere on the discussion. To select a layout type: From the Layout menu, select Layout Type.

**To change orientation**

- From the Layout menu, select Orientation.
- Select the direction you would like the view to display (top to bottom, bottom to top, right to left, or left to right).

[4] ibid.

### *To create a node*

- Click the button for the node you want to create. For example, you can click the Argument button to create a new argument. A new node appears near the center of the window.
- Double-click on the new node to open the Node Information window.
- Type information about this node in the Node Information fields.

To quickly add a node and relationship to your view, position the cursor directly over the node, press and hold the right mouse button, and select the relationship and node you want to add to your view. For example, you can create an argument node, then quickly create a qualifying assumption using the right mouse button.

### To delete a node

- Click on the node you want to delete.
- From the Edit menu, select Delete (Or press Ctrl+D).

### *To name a node*

- Select the node you want to name.
- Type the node name in the name field at the bottom of the window.

To view node information in a larger space, double-click on the node or click the Show Info button. To modify the name, description, or status of a node:

- Select the node you want to modify.
- Make your modifications in the information fields at the bottom of the window.

To get more information about the node than appears at the bottom of the window, double-click on the node or click the Show Info button.

### *To display the shortcut menu*

- Press and hold the right mouse button in the Discussion window (but not over a node).
- Select the node you want to create from the Shortcut menu.

To permanently display this menu on your window, select the dotted line at the top of the list.

### *To create a relationship*

- Click the Link button (this is the button with a straight line on it).
- Press and hold the left mouse button over the first node in the relationship.
- Drag the cursor to the destination node, then release the mouse button.

To quickly add a node and relationship to your view, position the cursor directly over the node, press and hold the right mouse button, and select the relationship and node you want to add to your view. For example, you can create an argument node, then quickly create a qualifying assumption using the right mouse button.

### *To delete a relationship*

- Click the Pointer button.
- Click the relationship you want to delete. From the Edit menu, select Delete (or press Ctrl-D).

### *To modify a relationship*

- Click the Pointer button.
- Select the relationship you want to modify.
- From the Edit menu, select Delete.
- Create a new relationship.

### *To display the legend*

From the Display menu, select Legend. This legend shows the possible relationships, and also the relationships you can use with the nodes. You can also see what kind of relationship each color represents. To see the possible nodes and relationships available from a node, position the cursor directly over the node, press and hold the right mouse button, and select the relationship and node you want to add to your view. For example, you can create an argument node, then quickly create a qualifying assumption using the right mouse button.

## OBJECT LINKING

### *To display the object link dialog for a node*

- Position the mouse pointer over a node, then press and hold the right mouse button. A floating menu appears.
- Select Show Object Links. The Object Link window displays the links for the node you selected.

### *To display the object link dialog for a topic*

- Position the mouse pointer anywhere on the Discussion Editor, but not over a node.
- Press and hold the right mouse button.
- Select Show Object Links. The Object Link window displays the links relevant to the topic.

### *To define an object link*

- Click the node you are linking from in the Discussion Editor.
- From the ObjectLink menu, select SetAsSource.
- Open the view that contains the object you want to link to your node.
- Select the object you want to link. For example, if you want to link Hypertext in the Hyper Document Editor, use the mouse to highlight the text you want to link.
- From the ObjectLink menu in the current view, select SetAsTarget.
- From the ObjectLink menu in either view, select FormLink.

To verify that the link was formed, press and hold your right mouse button over the node in the Discussion Editor window, then select Show Object Links. The Object Links window shows a list of all linked objects for that node.

# 5. ADM Logical Models

## 5.1 Software Development Model

### 5.1.1 Overview

The ADM is intended to support the design of software products as a process of evolving models. The ADM approaches the development of a software product as a process of gathering information and evolving models into a final form. For example, a requirements model leads to the creation of one or more specification models, and then to various implementation models, culminating with a final product "model." This process is similar to the way a sculptor chisels away at a block of marble while taking pictures of his work at various stages of refinement, until the statue is completed. Each new model represents a refinement of the previous model, and the final model is the resulting product of the development.

Earlier models are all retained within the ADM database, giving rise to the concept of a "version." Each model captures design decisions made at that particular stage in the life cycle of that model. Breaking development into models allows the developer to focus on concerns and express critical design decisions as they apply to that stage of development only. Thus it is clearer to someone reviewing the project why certain decisions were made -- they can see the logic behind decisions made within the context of the stage of development.

The ADM implements the model paradigm by providing tools capable of capturing the work done during particular stages. Hypertext documents and REMAP discussions, as well as ALE specifications and finally ALE packages (source code) are used to save pertinent design information for each model. Each time a Hypertext document, REMAP discussion or ALE specification is edited, a new version is created and represents a new, more refined model of the overall development effort.

### 5.1.2 Software Design Methodologies

When working on large software projects, it is important to adhere to some kind of design methodology which defines, for everyone involved, what procedures, practices and conventions will be followed. There are many different methodologies in existence, and one can easily create one's own methodology if a pre-existing one isn't suitable. A methodology provides a road map that the project should follow from start to finish, helping to define tasks that need to be accomplished, order the tasks in some logical hierarchy, and define clear milestones along the way.

The KBSA/ADM is flexible in the methodologies it supports. The ALE tool does rely on the (Rumbaugh) OMT diagramming method, associated with object oriented design, but there is no reason why some new tool might not be created which can support some other diagramming method. Methodologies are very important in the creation of a Microsoft Project work plan. Work plans define project tasks and resources (refer to the next section on the project model) which form the underlying project management structure for the ADM.

## 5.2 Project Model

The project model of the ADM centers on managing resources (users) and decomposing tasks among the resources. The goal of the project model is to tie together a communication infrastructure, versioning, and the supporting tools and make them work as part of one single collaborative environment.

A project in the KBSA/ADM consists of one or more users who create one or more sessions. Each session contains some number of tasks for the user to work on and complete. The ADM can manage more than one project, but no information is shared between projects, they are separate entities within the ObjectStore database.

Each project may have a different number of users with different names. Within a single project, all the products are visible between users. The "root" user is Super. It is Super's job to create a project and open an initial session, allowing a Microsoft Project work plan to be exported to it. Super is also in charge of creating and managing the other users on the project. Each user (possibly including Super) is assigned a number of tasks in the Microsoft Project work plan. These tasks dictate the work that each user will perform. Each task has a single resolution and one or more deliverables associated with it.

## 5.3 Communication Model

### 5.3.1. Overview

The ADM facilities are constructed as a collection of cooperating process. A central process, called the Session and Agenda Manager (SAM) manages the inter-tool communication. The responsibilities of SAM are:

- Manage the session-view paradigm
- Provide ADM login capability
- Provide support for topic creation, check-in, check-out and deletion
- Support tool registration and unregistration
- Support and manage collaboration within a team of developers

### 5.3.2. Messages

SAM interacts with each tool to perform some operations. Therefore, each tool is required to support some basic remote messages to make it compatible with the existing environment. These messages are:

1. **Tool Level Messages**
   - Registration
   - Unregistration
   - Suicide

2. **Topic Level Messages**
   - Create Topic
   - Delete Topic
   - Pre Check-in Topic
   - Pre Check-out Topic

3. **View Level Messages**
   - Create View
   - Delete View
   - Open View
   - Close View

4. **Object-Link Level Messages**
   - Source
   - Target
   - Form
   - Traverse

In addition to the above, each tool can define its own specific remote messages if needed.

The ADM supports both synchronous and asynchronous communication across tools. There are three levels of inter-tool communication in the ADM. These are the transportation, network and the physical levels.

### 5.3.3. Message Structure

In the ADM, each tool is connected to SAM. When a tool needs to send a message to another tool or SAM itself, it always sends it to the connected SAM first. SAM is responsible for routing the message to appropriate destinations. Each message consists of the following three basic fields:

1. **Return Address** (The source tool address, from where the message originated. The synchronous reply messages are initialized using this address)
2. **Send Address** (The name of the destination. This needs to be set for the tool specific messages by the source process of the message)
3. **Next Hop Address** (The connected process address through which the message needs to be routed. If the message is originating from a tool, this field is always set to SAM. If this message originates from SAM, then this is the same as the address of the destination tool itself.)

The messages can be modified to add more fields as and when required. The inter-tool communication is provided through a message-passing scheme. The return and hop addresses are set automatically as a part of the message constructor itself.

This is a very good design, as all the components involved are loosely coupled. Each component allows for extension while maintaining the consistency with existing features. New features can be added to the tool suite by adding new tools.

## 5.4 Collaboration Model

The ADM collaboration model is based on the notion that textual models allow us to describe concepts, goals and issues in great detail. The power of the written language allows us to capture large amounts of project information -- how decisions were arrived at, in what context, design discussions etc. Therefore, the ADM has the capability to structure, organize and manage this information.

ADM supports two types of collaboration models:

1. **Loosely Structured Descriptions (using HyperDocs):** As an example, consider the creation of an A-level specification document. Inputs to this task include transcripts of discussions with the client, user

manuals, functional descriptions of legacy code, white papers etc. The hyperdocument provides the capability to create and manage various hypertext based text documents. One strong feature of the hyperdocument is that it supports hyperlinks between heterogeneous design artifacts - a document can have links to discussion objects, code specification objects etc.

2. **Structured Descriptions (using REMAP-based discussions):** The objective of a REMAP discussion is to capture requirements and design discussions. Using REMAP, a user is able to replay the design history of a component and understand why and when different decisions were made. The REMAP allows a user to link issues, positions, arguments, assumptions, decisions etc. Typically, a discussion is associated with one or more work products. Controversial issues are recorded in the discussion and then related via object links to the pertinent portions of the deliverables. This way the user will be able to easily traverse and understand the design issues behind the various deliverable components associated with the project.

## 5.5 Product Model (ALE)

The product consists of the set of deliverables for all the tasks for a project after it is imported in the KBSA/ADM domain. Since KBSA/ADM supports the development of an object oriented system, it has the following features as a part of its product model:

- Well defined semantics
- Builds upon the common OO language models
- Consists of many OO editing transformations

It uses the underlying ARGO, which is an object oriented specification language consisting of concepts like class, attributes, relation, operation, and package. From this meta-model ALE defines several visual presentations including a class diagram, a package diagram and an ARGO view.

ARGO also consists of a data model including :

- Ability to maintain referential integrity on relationships
- Enforcement of cardinality constraints on attributes and relations
- Creation of appropriate destructors and copy constructors to realize desired semantics of composition relations
- Creation of initializers for all properties

Currently code generation is limited to generating header (.hpp) files and allowing the user to define

function bodies in the ALE environment which are then exported into a corresponding implementation (.cpp) file, both of which are outside the KBSA/ADM environment i.e. appear in a UNIX file system directory. This is because of the observation that code generation at unit level is cheap and of lesser value than being able to capture and describe all of the project processes to be covered at a higher level.
Object links facilitate coarse-grained impact analysis when requirements change and thus are in a better position to manage when and where to change the implementation.

## 5.6 Repository and Versioning Model

the role of the KBSA/ADM repository is to provide centralized persistence to design artifacts. The repository is built upon Object Design's ObjectStore OODBMS that provides a virtual memory model for object persistence.

The repository is organized into a two-tiered workspace:

- Project
- Session

The project workspace is a global workspace where objects are persisted and accessible to all users. The session workspace is a local workspace associated with a specific user.

The basic unit of versioning within the repository is a *topic*. The *topic* contains *First Class Objects (FCO)* or topics that are the smallest object that can be manipulated in the KBSA/ADM. Users of KBSA/ADM check-in and check-out topics (design objects) from the project workspace into their session workspace for transformation. The repository enforces a single writer, multiple reader protocol.

### 5.6.1 Topics definition

A topic is a deliverable you create in the course of completing a task. You use them to complete the deliverables assigned to the task in the methodology. For example, a business case may be created using various types of topics such discussions or specifications.

When you execute a task resolution, KBSA creates all topics you need to complete during this task. You can also create topics directly using options on the topics menu in SAM, and add them to a particular task. These added topics do not appear in the Microsoft Project work plan when the project manager imports it.

KBSA stores topics in the project repository, from which you check in and check out the topics you want to complete. Topics you create during a session do not exist in the project until you check them in to the

repository. Until you check a topic into the repository, other users cannot access it. You can have topics from different sessions open at the same time.

In KBSA there are four types of topics:

- Specification - a model consisting of multiple, related packages. You document a specification using the package diagram view.
- Package - a collection of classes and their relationships. A package is documented using class diagrams.
- Discussion - a record of a meeting, conversation, or other communication in which issues and arguments relevant to your project are discussed. You document discussions using the Discussion view.
- Hypertext document - a free-form text file with links to related documents or diagrams.

### 5.6.2 Topic Versions

A version of a topic is a record of a topic as it existed when you checked it in to the project repository. When you create a topic, KBSA automatically assigns it the version number 1. Each time you check a topic out of the repository, KBSA increments the version number by 1.

You can check out only one version of a topic in one session. However, you can check out different versions of a topic in different sessions. This allows you to compare current versions of a topic with earlier iterations.

You can check out the most recent version of a topic in write or read only mode. You can check out prior versions in read only mode. You can only check out a topic in write mode in one session at a time.

You can have multiple views for a topic. For example, you can open two different discussion views for the same topic. Items you add to one view (a new argument, for example) appear simultaneously in the other view. However, layout changes made in one session are not reflected in other views. This allows you to compare different layouts of the same material. You can have different views of the same topic open at the same time. You can also have views from different sessions open at the same time.

When you check a topic into the repository, KBSA does not preserve views. When a new version of a work product is created, the ADM uses the agenda mechanism of IPSE to inform all developers using the older versions that the work product currently in their sessions is no longer the latest version. The notification appears as a critic resolution task and offers as a resolution to check-in the current work product and check-out the latest version. By using the agenda mechanism, the ADM gives a user the

flexibility of knowing as soon as possible that they are working with an older version, but allows the developer to delay integration of the new version until ready.

The versioning system facilitates exploration of the design space by allowing alternative versions of a work product to be created. Object links are used to connect the discussion to the appropriate portions of the designs (in either version). In this way differences between the versions can be highlighted and explained. When one version is selected over another the rationale behind the decisions can be captured as the conclusion of the discussion.

## 5.7 Substrate Model

### 5.7.1. How the tools work together

The tools work together based on the session view paradigm, which provides users with a perception of the ADM as a seamless development environment, rather than a set of loosely coupled independent tools. The three key layers in the overall ADM session-view architecture are shown in the figure below:

The three layers are:

1. Session View Layer: This shows the user views, which are grouped into sessions, based on some logical grouping of the user's tasks.
2. Tool Layer: This contains SAM, the various ADM tools, the Requirements Acquisition Support Environment (RASE).
3. Repository Layer: This consists of the back end facilities for managing the persistent design objects.

All tools are required to register with SAM when they start up. The tools let SAM know about the topic types and the view types that they support. The SAM menus are dynamically configured at run-time to reflect this information; thus, SAM is easily extended to support any new topic type.

After this, SAM presents the user with a view of his/her session. The session is a logical grouping of some of the tasks that the user has to perform. In this session, most of the tasks have output deliverables. These deliverables are topics, and the operations that have to be performed on the topics can be broadly divided into three parts: (1) Topic Creation and Deletion (2) View Creation and Deletion (3) Check-In/Check-Out. SAM doesn't know how to create and manage these topics and their views. So, items (1) and (2) are managed solely by the tool that supports the topic. All SAM does is provide the tool with the requisite topic and view level messages, as described in detail in section 5.3.2. However, it is SAM that manages (3). Obviously, all topic check-in and check-out are best done by a single, central agent. During check-in and check-out, SAM provides the tools with *pre* and *post* messages, which lets them know of SAM's intentions to check in or check out topics; the tools can perform some specific actions (if they need to) when they receive these messages.

### 5.7.2. Addition of new tools and topics

One of the principal goals of the KBSA/ADM project is to provide an open-ended architecture that enables the user to customize the environment and extend it. This section briefly describes the activities involved in adding a new tool to the ADM environment:

1. **Defining the tool functionality**: What exactly does the new tool do? Is it going to introduce new topic types into the ADM environment? Adding a new topic is not always necessary, as it is possible to add new functionality without adding new topics. One can present the information in the existing database itself in some new form, or one can add a tool that adds some product management capabilities to the ADM etc. Whatever the case may be, defining the tool functionality is the first step in adding a new tool to the existing tool suite.

2. **Defining the User Interface**: The next step is to design the tool interface. This doesn't mean drawing the UI in terms of GUI components alone; it means one has to carefully think about how the user will interact with the software, what messages are to be generated for each of those interactions, and how to respond to each of those messages.

3. **Providing persistency**: The ADM environment gives the user lots of functions for providing persistency in the database. The developer of a new tool has to decide what objects he needs to make persistent. The views are best designed if they are compliant with the ADM's model-view concept; this concept says that the views for topics are always dynamically generated, and are not persistent. Only the topics themselves are persistent.

4. **Enabling the inter-tool communication**: Once the tool is working in its standalone mode, the developer can add inter-tool communication facilities to it, using the frameworks provided. Here, the developer has to ensure that the tool registers itself with the environment during start-up, responds to all the required messages from the other tools etc. This is the phase that really links the tool into the ADM environment.

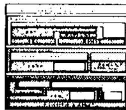
The specific details on how to achieve each of these steps are outlined in the following section on individual tool architecture.

## 6. ADM Architecture - Physical Models

### 6.1. Individual Tool Architecture

Each ADM tool is built on top of a number of object oriented frameworks. A framework is a set of collaborating classes to provide a domain specific capability. The main advantages of the frameworks include software re-usability (higher productivity, better portability, and higher reliability). Also, uniform interface standards are easily enforced this way. The structure of an individual ADM tool is shown below:

Each tool has three basic layers, and each layer has some frameworks associated with it. The three layers



are:

1. Presentation Layer: This GUI layer is responsible for the visualization and manipulation of work objects on the screen
2. Message Layer: This layer handles all the inter-tool communication, and provides loose-coupling between the various tools in the suite.
3. Repository Layer: This layer is concerned with the persistency and version management of the various work objects. It also provides the object linking capabilities among the objects.

The logical model for a tool (as outlined in section 5) is realized by the suitable interaction and data flow between these three layers.

## 6.2. ADM Frameworks

The process for creating an ADM compatible tool requires that the developer define the tool's purpose, its set of work objects, and the features it needs to support. The tool developer selects frameworks from what the ADM provides, and specializes and integrates them to meet the tool's requirements. Thus, the efforts of the developer are mainly focused on integrating the various frameworks, rather than developing the features from scratch.

### 6.2.1. Presentation Layer Frameworks

#### 6.2.1.1 Galaxy

The ADM uses several presentation layer frameworks. The most important of these is Galaxy, a commercial platform independent framework for developing GUI applications provided by Visix Software Inc. Galaxy is used in the ADM to provide all of the GUI services such as the SAM window and the various tool interfaces. The Galaxy/C++ library consists of *managers*. A manager is a collection of collaborating C++ classes for providing a specific capability. Different managers include: dialog manager, menu manager, button manager, list manager, text manager, memory manager, etc. Using Galaxy takes a lot of work out of designing the user interfaces because it comes with VRE, a visual resource editor which reduces the amount of time involved in designing the various user interfaces and making changes to them during development. The resources (dialogs, message boxes, icons, etc.) are created in vre and saved in a separate file. The program that utilizes the resources accesses the .vre file to get to them, they are not compiled into the binary executable of the program. This does two things:

1. It keeps the size of the executable program smaller by loading GUI components dynamically at runtime

2. It allows dialogs to be modified during development without necessarily having to recompile the code each time

One drawback of Galaxy is that there is a very high learning curve involved in using it since it is a very extensive and flexible API, capable of supporting many complex activities and it has an enormous number of different functions and features. However, only a small subset of the Galaxy functionality has actually been used in the KBSA/ADM, which somewhat mitigates that high learning curve.

#### 6.2.1.2 AGL

AGL stands for Aesthetic Graph Layout. This framework provides automatic graph layout capabilities such as node positioning, moving and resizing, edge routing, nested graphs, and multiple layouts. Andersen Consulting created this framework and encapsulated within GEF (the Graph Editor Framework -- refer to section 6.2.1.4) to provide transparent API of diagrammatic interface programming constructs for use in tool development. The results of this framework are clearly visible when creating REMAP discussions or when designing ALE package and class diagrams.

#### 6.2.1.3 Scribbles

Scribbles is another framework created by Andersen and encapsulated within GEF (§6.2.1.4). Scribbles provides the capability to manage (create and control) the graphical entities such as rectangles and lines. It extends the pre-existing Galaxy graphical features, providing an API, that along with AGL, makes up the backbone of GEF.

#### 6.2.1.4 GEF

GEF, the Graph Editor Framework provides a powerful and customizable diagrammatic interface. GEF supports automatic visualization capabilities:

- Customizing dialogs with default menu items, buttons, and interactors
- Multiple automatic layout types: Tree, hierarchy, network, and manual layout
- Nested diagrams, automatic node positioning, and link routing
- Popup menu support at every object level
- Automatic decoration and identification on the diagram objects
- Enforcement of standardization in diagrammatic user interface

The success of the GEF framework can be experienced through using ALE to create package and class diagrams, and in the REMAP discussion editor while creating discussions. Many of the menu items in both of these tools rely heavily on GEF functionality.

### 6.2.2. Message Layer Frameworks

The two frameworks available for developing the message layer of the tool are:

**1. Extended Model View Framework (XMVF)** is a model-view controller framework, similar to those found in SmallTalk applications. The key objectives of this framework are:

- To establish and maintain relationships between the ADM views and back end model objects (topics)
- To provide a consistent mechanism for defining messages between the views and the model of the system, while hiding the details of transaction control and management.

XMVF implements an ObjectStore independent meta-model of topics and first class objects on top of ObjectStore's concepts of os_configuration and object. In cooperation with object-linking, it also implements notions of relationship within a topic (using persistent pointers) and across topics (via object links). XMVF also provides the basic communication infrastructure to perform actions on these objects from views and deliver these changes to the views in a synchronized manner.

XMVF satisfies the following requirements:

1. Provide a KBSA-specific meta-model for user objects managed by the KBSA Repository that is fitted to the user model for KBSA and that is not directly dependent on the database implementation.
2. Provide object uniqueness that is stronger than address uniqueness in pointers
3. Interface with the actual database to deal with transaction handling, so that it is transparent to the tool developer
4. Provide a common platform for requesting, executing and reporting the results of actions against the contents of the repository
5. Provide a basic Observer pattern within the repository as a tool for tool developers.

**2. Remote Operations (RO):** RO provides tool-to-tool communication across a network, and is

responsible for the following functionality:

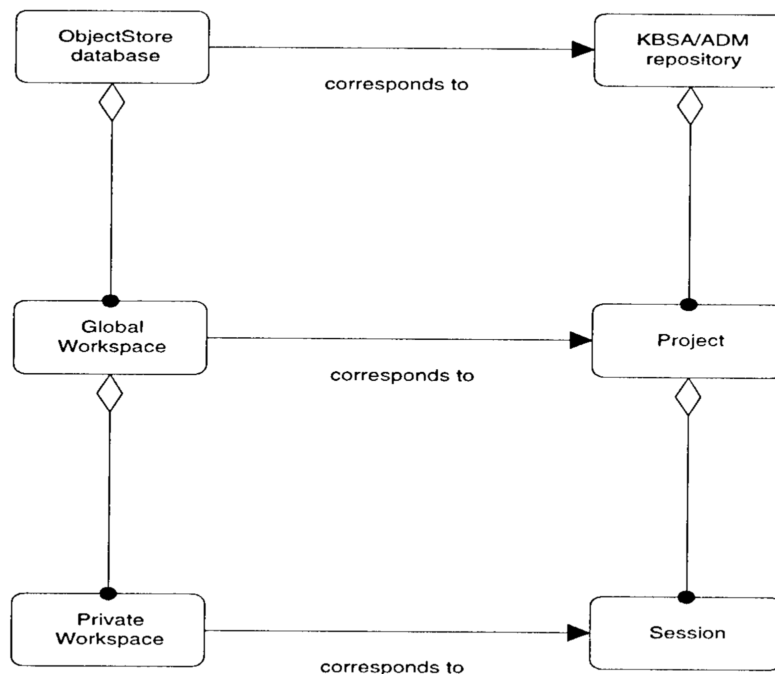
- Connectivity to existing tools through ports
- Synchronous and Asynchronous communication protocols
- Message-based communication
- Spawning and terminating tool processes.

There are different kinds of remote operations, such as IPSE Tool Commands, Stored Task Execution, Proxy Synchronization, Remote Stand-Alone Task Execution, SubTask Execution, COTS Tool Integration and Object Linking. The connectivity framework provided by RO hides from the user all the details of socket programming.

### 6.2.3. Repository Layer Frameworks

The Role of the KBSA/ADM is to provide centralized persistence to design artifacts. Persistence and Versioning management layer is built on top of the existing OODBMS ObjectStore and has the following features :

- Version Management of objects both in projects and sessions.
- Supports full check-in/check-out facility
- provides utilities to create databases, projects, users and sessions



Objectstore to PVM Concept

As shown in the figure above the Objectstore database corresponds to the KBSA/ADM repository which is chosen to be a file based repository.

The global workspace concept from Objectstore maps to the project in the KBSA/ADM domain. The project consists of a set of of KBSA/ADM tasks and topics. So topics and tasks within that project are not "visible" to other projects. Sessions are created by a user with the SAM component of the KBSA/ADM. The private workspace represented by the session contains project tasks and topics checked out from the Project, to topics created by the user and not yet checked into the project.

### 6.2.3.1 Concept

The PVM capabilities within the KBSA/ADM wrap around the corresponding capabilities of Objectstore as follows:

- Ability to persist objects
- Use and management of user and shared work space
- Version control
- Transaction and messaging support

The first three objectives are achieved automatically via built-in Objectstore functionality. Transaction management and messaging support is customized for the KBSA/ADM application domain which enables successful decoupling of the tool developer's need to understand Objectstore technical details including the details of the transaction management. All the routine transaction semantics ( start, commit/rollback ) are hidden from the tool developer.

The concept of the request is introduced at the XMVF layer level which ties in with the PVM layer closely. Each request has a corresponding *doIt()* method that performs the actual request. However it is not directly invoked by the tool developer but called by the XMVF layer's execute() method which handles all the transaction setup and invokes the same.

The typical process of KBSA/ADM transaction management can be described as :

1. Communication between a tool and the repository occurs through the PVM layer using "request"
2. Requests are created using tool specific attribute values and then "executed"

3. The "execute" method initiates Objectstore transactions, invokes the corresponding doIt() method, and post-processes after the doIt() returns. Tool developers need only concern themselves with these pre- and post-transaction conditions relative to the processing of the specific messages.

#### 6.2.3.2 PVM Requests

There are two types of requests :

- Requests which are exclusively for the KBSA/ADM technical architecture components
- Requests which are available for sub-classing by tool developers

Some of the typical requests that will be used by a tool developer are

- PV_DELETE_TOPIC_REQUEST
- PV_CHECK_OUT_REQUEST
- PV_CHECK_OUT_FOR_WRITE_REQUEST
- PV_CHECK_OUT_FOR_READ_REQUEST

## 7.0 Adding a New Tool to the ADM

The ADM environment currently has tools for planning and controlling the execution of a project's tasks, generating project design documentation such as Hyperdocs and REMAP discussions. It also has a facility for specifying code components, resulting in the generation of header files. A real-world software development process definitely needs these project planning, and topic-development tools. In addition, it also needs some facilities for managing the entire software development product baseline, such as:

- Support for module and test plan topic types
- Support for managing and browsing software products
- Support for spawning documents: Manuals, test plans, reviews etc.
- Support for the Software Integration process: Code builds and document builds.

One important issue concerning the ADM implementation is just how difficult is it to add a new tool to the ADM federation? We decided to attempt to add a new tool to the ADM that extends the ADM's model of the software development process to include the facilities mentioned above. We call this new tool the Project Archival and Report Tool (PART).

### 7.1 The PART Functional Model

PART introduces the concept of a **Project Knowledge Structure** (PKS) in an attempt to satisfy the above requirements. A PKS is essentially the MS-project plan, augmented with some additional information. Hence, it contains information such as the list of tasks, their inter-dependencies, the resources allocated to the tasks, and their deliverables. A PKS is shown in Figure 1. Each box in that figure corresponds to the notion of a task in the ADM. Hence, the tree represents inter-dependencies of the tasks between themselves. PART will display a graphical representation of the PKS in its window. The PKS can be shown either for the entire project (global view), or for a single session within the project (distributed view). This is discussed in more detail in a following section.

What is the idea behind showing the PKS to the user? The intention is to provide a graphical interface to the project itself--the user can browse a project's tasks and deliverables in an easy and convenient manner. PART concept also supports **visitors** and **builders**; visitors allow the user to browse the PKS according to some search pattern, and the builders allow the users to generate documents and build code.

Visitors present different kinds of views to different kinds of users. For example, a *general's tour visitor* might be targeted towards someone in the higher management, who wants to see only the broad divisions

of the project and the major design issues, without getting into the details. A *team leader visitor* might be designed such that it presents a lot of details about a particular process within a project.

A **build** refers to a process by which deliverables are selectively extracted from the repository to form a meaningful collection. With this facility, developers will be able to generate execution images for modules, manual pages for all sub-modules for a process, and test plans for a multi-module component.

PART includes two new topic types to support these goals. Specifically, PART supports a **module topic type** and a **testplan topic type**, and the associated views for them. A module topic represents a software module with a header file and an implementation file, and a testplan topic represents a software test plan.

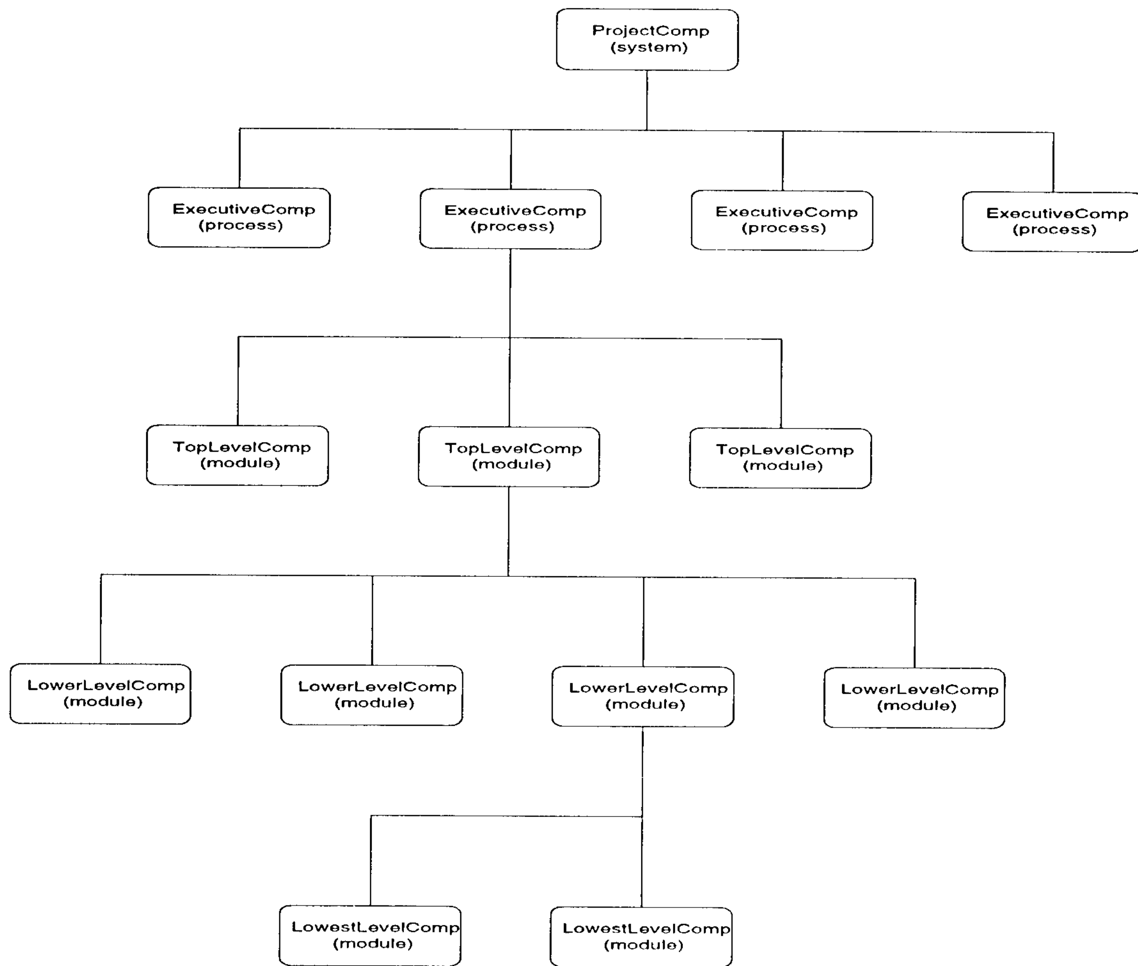



Figure 13: **Project Knowledge Structure**

## 7.2 Conceptual Model for PART

The aim of the conceptual model is to illustrate the relation between the ADM's notions of tasks and deliverables to PART's notion of components and products.

The following diagram shows the connection between the ADM world and the PART world:

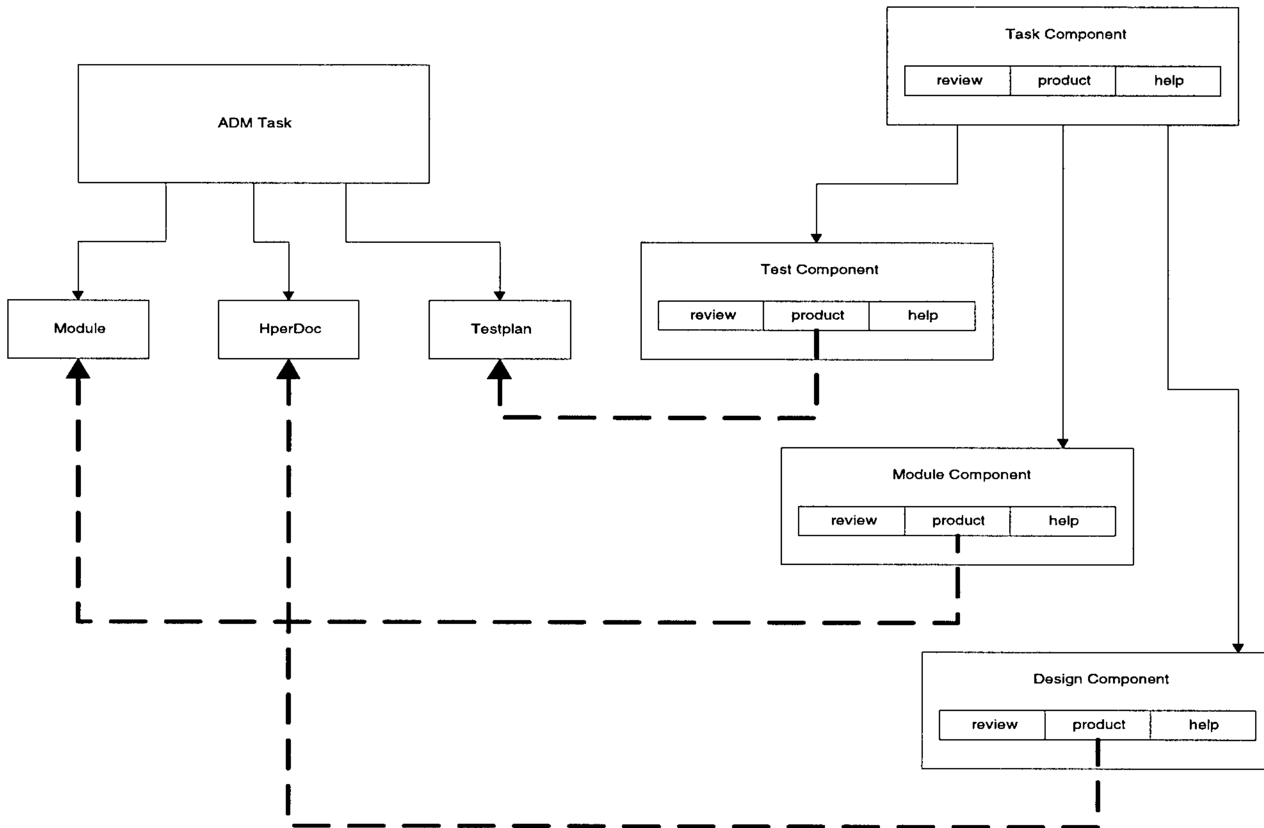



Figure 14: **Conceptual Model**

The dotted lines in the above figure show the runtime pointers made from the component structure to the actual topics created by the ADM.

During our design efforts we tried to focus in on what functionality PART should encompass and what its boundaries are. Some of the assumptions made are:

- PART visitors are able to traverse the whole database file
- A Module topic always consists of one header file (.hpp) and one implementation file (.cpp),

stored in the database

- PART does not support any kind of role mechanism (e.g. the hierarchy of Project leader, team leaders, developers) but this would be a natural and useful extension.
- A Testplan topic is associated with a single module.
- The component structure is based on the notion that a user will want to check out the topics that are created or modified by a task. Update deliverables and output deliverables are considered to be equivalent for the purposes of the component structure. Input deliverables are considered only for building up the task dependency hierarchy; beyond that, the component structure does not deal with input deliverables.

## 7.3 PART Basic Types

### 7.3.1 Introduction to Module and Testplan Topics

The Module and TestPlan topics are two new topics that are designed to store, as part of the ADM database for a project, the source code and the unit test plan for a software component product. The Module topic consists of a header file and an implementation file (.hpp and .cpp files). The view for a module topic allows the developer to specify the names of the two files, and examine both files concurrently. The current interface is read only -- if the developer intends to do work on these files they must be extracted from the database (by saving them to a directory) and then edited outside of the ADM environment. When the developer finishes editing the files, he then opens them in the ADM module view dialog and saves them in the database.

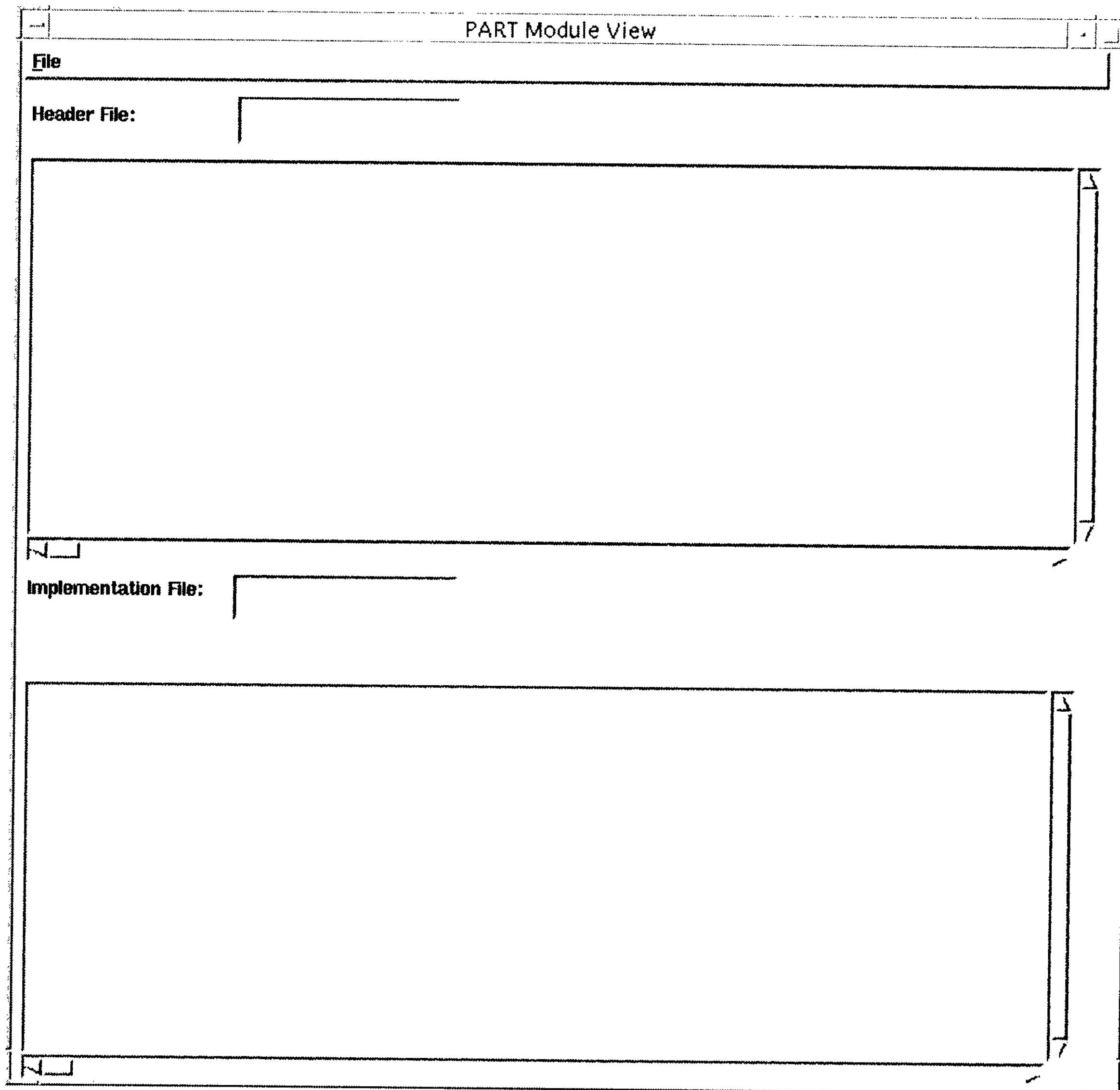
The TestPlan topic dialog allows the user to enter information relating to the testing of a component. This information is then saved in the database and can be referenced or updated whenever necessary.

Screen views of the dialogs for these topics are shown in the next section.

### 7.3.2 Screen Shots



View for the Test Plan Topic




View for PART's Module Topic

### 7.3.3 Implementation Strategy

The steps we have followed to implement PART is as follows:

- Conduct discussions to clarify our notions of a testplan and module topic
- Formulate views for the new topics
- Write code for tool Registration, Unregistration, Suicide

- Write code to create a new testplan topic from SM_TOPIC_TYPE with no data members in it
- Write code to handle messages for
  - Create
  - Delete
  - Checkin
  - Checkout

- Use Galaxy to create a view for the testplan topic
- Write code to display the view. At this stage, the view doesn't depend on what is in the testplan topic; it is a static view.
- Use Galaxy to design a view for the Module topic
- Write the code to provide all the basic functionality for the module topic also (create, check-in, checkout, delete, display a different static view)
- Write code to make the view dependent on the topic data itself

Advantages of using the ADM environment for tool development

The ADM environment provides the tool developer with many convenient features, which are otherwise time-consuming to develop. Some of these features that the developer gets for "free" are:

> Topic Level Features: Persistent Topic Creation and Management support, check-in and check-out facilities with version control, view creation and management support, object linking etc.
>
> Tool-to-Tool communication
>
> Multiple session handling capabilities
>
> Multiple user handling capabilities

These features can be incorporated into the new tool with small effort by making use of the various ADM frameworks.

## 7.4 PART Project Knowledge Structure

### 7.4.1 Introduction

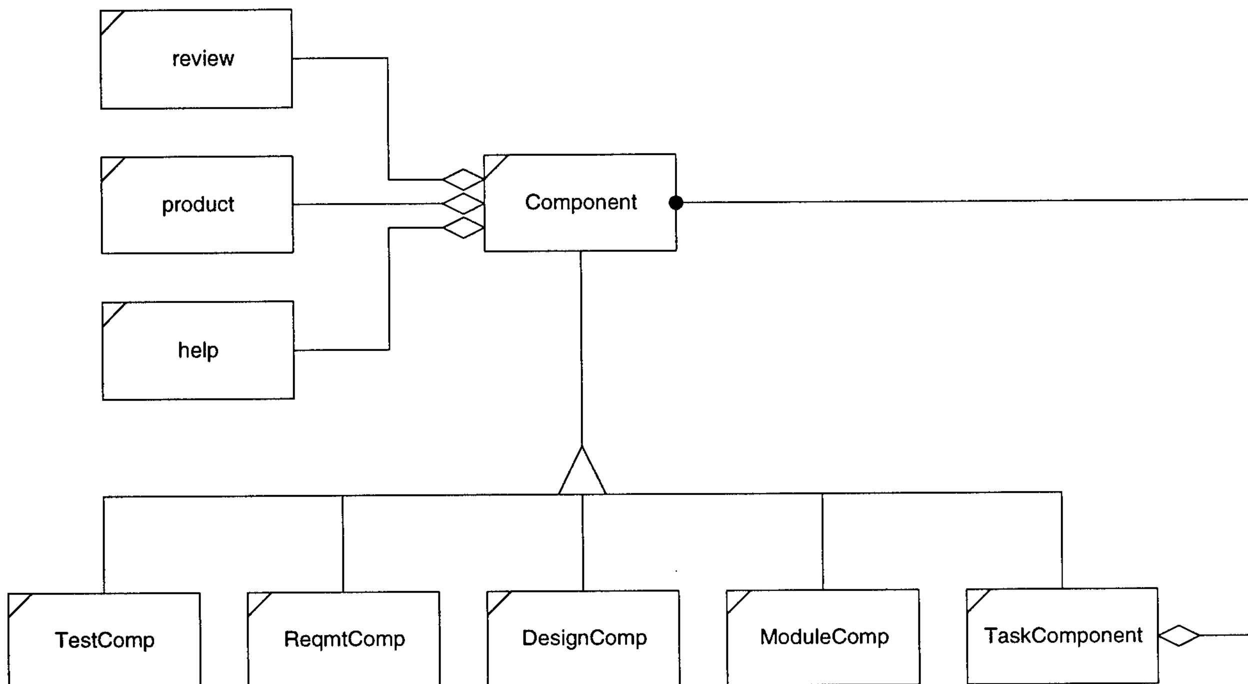
The PART dialog presents the user with a graphical representation of the task hierarchy, and the status of each of the task--how many of the deliverables have been created already, and how many are being currently modified. PART also invokes the views for these various topics, and thus serves as an object browser for the project tasks. In this sense, the PART dialog is not a *view* according to the model-view framework. There is no *model* behind the PART dialog, all it does is represent the collection of information in a graphical way.

### 7.4.2 Object Model

#### 7.4.2.1 Component Structure

The OMT diagram on the next page specifies the classes involved and their relationships for implementing the project knowledge structure. The *Task Component* is an aggregation for the whole Project Knowledge Structure. This maintains a list of pointers to objects of type Component, each of which can either be a *TestComp*, *ReqmtComp*, *DesignComp*, *ModuleComp* or a *TaskComponent* itself, which gives the structure the ability to add an additional level to the Project Knowledge Structure. Thus *TestComp*, *ReqmtComp*, *DesignComp*, and *ModuleComp* are the leaf nodes of the Project Knowledge Structure.

Each Component owns three objects, which are *review*, *product* and *help*. The *product* object contains the object identifier *productOId* to its persistent topic object. The review component contains any review comments for the topic under *product* and the *help* class contains help text for the same topic.

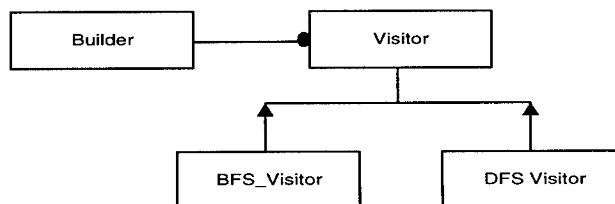



#### 7.4.2.2 Persistent Component Structure

The structure described above takes care of the recursive nature of the PKS. We require PKS to be persistent and versionable, since it contains references to persistent topic object from the KBSA framework. We integrate this Component structure with the KBSA PVM layer.

### 7.4.3 Visitors

Visitors implement the functionality of code-build and document-build for PKS. Code-build is meant for




topics that are referenced by the ModuleComp objects at various levels of the PKS.

The OMT diagram, above, describes the design of the object model for the builder/visitor concept. We currently define two types of visitors characterized by their search pattern: Breadth First Search Visitor and Depth First Search Visitor, each of which is a Visitor object. The builder has access to a visitor

object, which can be instantiated and linked to the data member of the builder at run-time. The visitor functionality will be invoked with the help of menu-items in the Global PKS window view.

*Sample Build process description for Test-Plan build with a Depth first visitor strategy for the product information with destination as a file:*

- The user can specify graphically (with the help of mouse clicks) the branch(es) to be traversed from the PKS. By default the whole forest will be considered as selected. The Component structure will be traversed as follows:
- For each tree root:
    - For all the TestComponent objects in the list data member of the Task Component object, node:
        - get the oid of the topic from the product data-member
        - get the relevant information stored in the topic with the help of topic object's member methods
        - Append to a temporary file
    - Visit the next node in the tree according to the classic Depth first tree traversal
- Copy the temporary file to the file the name of which was obtained from the user

### 7.4.4 More Screen Designs

The Visit menu item will allow one selection to be made from each of the following grouping: BFS/DFS, CODE/TEST/DOCS, PRODUCT/HELP/REVIEW. Once the user has made the selection he can use the Build menu item to select the destination of the build. This will consist of either a temporary file generated as Hyperdocument and shown on the screen or exporting the built information to a user specified file, determined with the help of a File browser window.

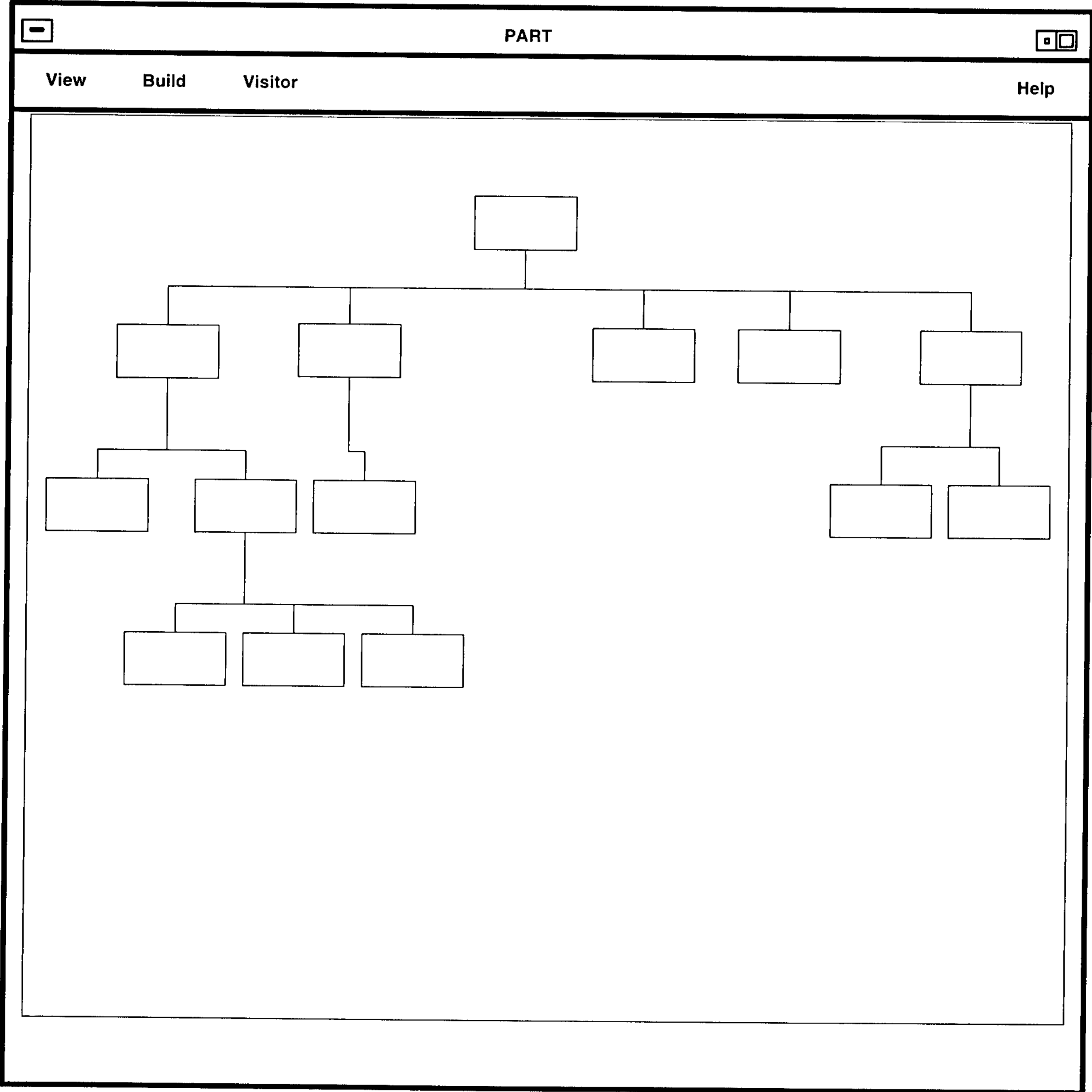
The following dialog shows checkboxes. From BFS and DFS, only one can be selected at a time. From CODE, TEST and DOCS, only one can be selected at a time:

**Visit** **Build**

BFS ◇
DFS ◇
CODE ◇
TEST ◇
DOCS ◇
Product ◇
Help ◇
Review
To Screen ◇
To File ◇

PART Dialog Choices

The screen design below shows the PART navigation screen. This is intended to give the tool user an overall view of the project for purposes of extracting information or initiating a code or document build.




PART Main Dialog

### 7.4.5. Implementation Strategy

This section deals with the actual implementation phases of PART including observations about the ADM made during the process.

#### 7.4.5.1 Design and Phase I

The design phase lasted several weeks. The result of this phase, a detailed design document, can be found in Appendix A. The previous subsections have summarized only the functionality-related aspects of that design document in order to give the reader a flavor for PART's functionality without detailing implementation issues.

We used the majority of the design time gaining an in-depth understanding of the KBSA/ADM architecture at the code level in order to get an idea of the different integration strategies for PART. The major functionality of PART, the PKS navigator, is not a KBSA topic in the strictest sense since it attempts to navigate the IPSE structure and view an entire project. This type of functionality transcends the usual tool functionality supported by the ADM. Thus, in order to design such a tool, we had to get advice from the Andersen team. Mainly this advice consisted of us trying to find out what functionality the ADM can and can not support. We also had to make some compromises on our design due to the time constraints, mainly a scaling back of proposed user-interface functionality.

The implementation of Phase I lasted three days. We used design document we wrote as a guide and began creating a tool that would provide the desired functionality. Bhaskar Naidu from the Andersen team assisted us on site. During this phase we achieved two important milestones. First, we were able to create the underlying message handling structure for PART, the most important piece. Second, we successfully implemented two new topics, the Module topic and the TestPlan topic. This phase of the implementation went quite smoothly. We found that the ADM can accept new topic types quite easily. We were able to complete everything except the actual user-interface functionality for the dialogs that we designed -- we did not finish the code that controls the passing of user entered information from the dialog to the database and vice versa. We considered this functionality to be less critical since we knew that it was easy to add. Our efforts concentrated more on trying to see how difficult adding a new tool would be, not on getting all the particulars of the user interface completed. This phase was hampered by some equipment failure. Had it not been for that, we would have succeeded in making both topics fully functional.

#### 7.4.5.2 Phase II

Following the Technical Interchange Meeting held at Syracuse University in early May, Bhaskar returned for two days to help us continue the implementation of PART. This second phase focused on creating the interface and functionality for the Project Knowledge Structure viewer. The PKS viewer is really just another new topic, added to the two existing ones, plus some additional functionality not normally associated with the role of a topic. We were able to build on to the software created during phase I. We added a dialog that displays the PKS for the user and added some of the functionality for displaying a tree structure with linked rectangles (signfying tasks and their dependencies). There was not enough time to make the PKS viewer fully functional. The work done serves as a proof of concept: None of the visitor or display (i.e. double-clicking on a task to view a deliverable or other information) features were realized. Only tasks added via the SAM menu appear in the PKS navigator dialog, the integration with IPSE proved to be non-trivial. In order to have full functionality for PART integration with IPSE would be necessary.

### 7.5 Problems Encountered

We met with mixed success in implementing PART. This mixed success had more to do with time constraints than it did with problems integrating the tool into the ADM. We successfully created two new topic types, and had some success with getting the user interfaces working fully. The module topic got to the point of being able to read in the text of a source file from the UNIX file system and display the contents of the file in one of the windows of the topic's dialog. Although we did not get to the point of adding all the desired functionality for our two new topics that we wanted, we were successful in writing all the code for the underlying operations that a topic or tool must handle. That means that about nine tenths of our implementation of the topics got completed, and if time had permitted, there is no reason why they could not have been finished. In addition, we were able to complete some of the project navigation functionality for PART, including a demonstration/proof of concept PKS view of tasks added from within SAM.

It took us a long time to fully define our concept of what PART could do and should do. It also took us time to fully understand the architecture of the ADM. As a result, we could not complete the actual implementation, but were satisfied that the PART tool could be fully implemented, with a few person weeks of effort. Through our efforts, we learned about the inner workings of the ADM, especially about how tools register themselves and communicate with SAM, and what messages are passed between

processes. This deep understanding of the ADM can only be reached by those who actually write code for a new tool and attempt to integrate it. So although the functionality of PART was less than we had hoped for, the process was of value to the overall evaluation of the ADM. This experience gave us insight into the framework structures that compose the ADM, as well as the relative ease with which a new tool can be integrated.

## 7.6 Results and Conclusions

Overall, our efforts toward designing and implementing PART met with positive results. The following checklist of goals illustrates our progress:

**Phase I**

| Design of Part | Completed |
|---|---|
| Code for tool registration, unregistration, and suicide | Completed |
| Code for TestPlan topic | 50% complete |
| Code for Module topic | More than 50% complete |
| Topic creation, deletion, check-in, check-out | Completed |
| Use VRE to create dialogs for the new topics | Completed |

**Phase II**

| Code for PKS topic - started in Phase I | Complete |
|---|---|
| Use VRE to create PKS navigator dialog | Complete |
| Code for enabling visitors | Just begun |
| Code that ties PART to IPSE to replect an exported work plan | Not attempted |
| Code to support user interface features such as double-clicking to display help, review, and | Not attempted |

There is a learning curve involved in adding a tool to the ADM, but there are no huge hindrances involved in the process. During phase I we were able to create the interfaces for two new topics and implement most of the functionality for both except for the views. In phase II, we made a lot of progress in getting the PKS navigator up and running with demonstrable functionality.

In the process of our implementation we found that all of the hooks for fully integrating a new tool exist. There is no reason why such a tool could not be every bit as integrated into the ADM as ALE and RASE are. This ease of extendibility above all is the really important point to elicit when discussing what the KBSA/ADM provides. It means that the ADM can be easily extended to add nearly any kind of CASE tool that might be desirable to have. PART itself is one example of such a tool.

# 8. RABS Maintenance Activity

This section is divided into two fairly long parts. The first describes a Repository and Build System (RABS) used as a vehicle for evaluation of the ADM. The second part describes our use of the ADM for maintenance activities on RABS and conclusions about the effectiveness of each of the ADM's supported tools.

## 8.1. RABS Overview

We wanted to evaluate the ADM by actually using it in the way it was intended to be used--on an actual software project. Although it might have been more consistent with the ADM development model to start a new software design project from scratch, we did not have enough time to begin one that would be a non-trivial exercise of the ADM. Our compromise was to perform a maintenance activity upon a pre-existing software system. The project we chose, Repository and Build System (RABS), needed some further refinement before it could be released for public consumption. This refinement consisted of fixing several bugs and adding some new functionality to the system. This maintenance activity was deemed small enough to be successfully completed in the time available, as well as complex enough to benefit from the kind of project management that the ADM provides. A description of the RABS architecture proceeds, followed by a detailed account of our evaluation strategy and the results of the evaluation.

## 8.2. RABS Logical Model

RABS consists of more than 20,000 lines of source code, not counting compiler library. It was developed in six intense weeks in the Fall of 1996 as a class project for the Software Studio course, part of the Computer Engineering Program at Syracuse University.

RABS is a tool designed to help developers manage a source code repository. It has facilities for:

1. Viewing the repository directory structure
2. Searching for components (each component is a list of files) by name or keyword
3. View code manual pages, maintenance pages, and source code
4. Insert, delete, update, and extract components and individual files

Its primary goal is to store and extract, for reuse, software components, each of which may consist of many source code files. Components may be accessed by name and may be extracted and compiled as entities. A user may view a list of all components in the repository, may search for a component by name or keyword, or view individual files to understand their interfaces and implementations. Finally, the user can ask that a set of components be compiled and linked to form an execution image.

## 8.3. RABS Architecture

The RABS has two separate executables -- the repository manager and the build manager. Each of these uses the services of modules:

1. Database subsystem
2. Directory subsystem
3. Viewer subsystem

The repository manager supports both tiled text window and command line interfaces. Its primary responsibility is to save multiple versions of files and components and disclose information about them to a software developer. The build manager also supports both tiled text window and command line interfaces and also supports scripts for controlling complex software builds. A script describes the components and files used to create one or more execution images and the command strings necessary to activate a compiler with whatever options are appropriate for a specific build. Scripts can be commented to maintain an audit trail for the build. For example, with two commands the build manager can unpack a script and source code and build both itself and the repository manager.

## 8.4 RABS Design and Implementation

### Database Management System

The DBMS supports the management of linked fixed length records to store a file of arbitrary size. The software provides for insertion, update, deletion, and extraction of a file designated by a numerical key. Each file is divided into a series of fixed length records with some linking and directory information included in a header. The tasks for this subsystem are outlined as follows:

- Add file in database
- Delete file from database
- Extract file from database
- Ensure identical files

Database Directory Component

Provides a two-tiered directory structure. The top level is a lexicographically sorted listing of each of the components in the database. The second tier lists each of the files that make up a component (build files and documentation files), also in lexicographic order. This component provides a list in lexicographic order of all files in the database matching a pattern. Each file in each component is associated with a numerical key pointing to its records in the database. Directory software provides the capability to search by component name, file name, or keyword for a matching component.

The specific tasks for this subsystem are each assigned a command line option which gives it the functionality of manipulating the file and/or components in the database in a via a text based GUI or from the command line.

The storage of all the files is done in a single database file, which enable the user to be able to easily move the database file(s) around the operating system, make backups, etc. When inactive, the database is stored in the following format:

| File header | Database File | Index File | Comp Index | Viewer Config |
|---|---|---|---|---|

The index file is a temporary file generated when the RABS session is in progress. It is written at the end of the database file when the RABS system exits whether or not execution completed successfully. The index file must always be latest version and consistent with the database file, since the records in the database file are accessed through references in the index file.

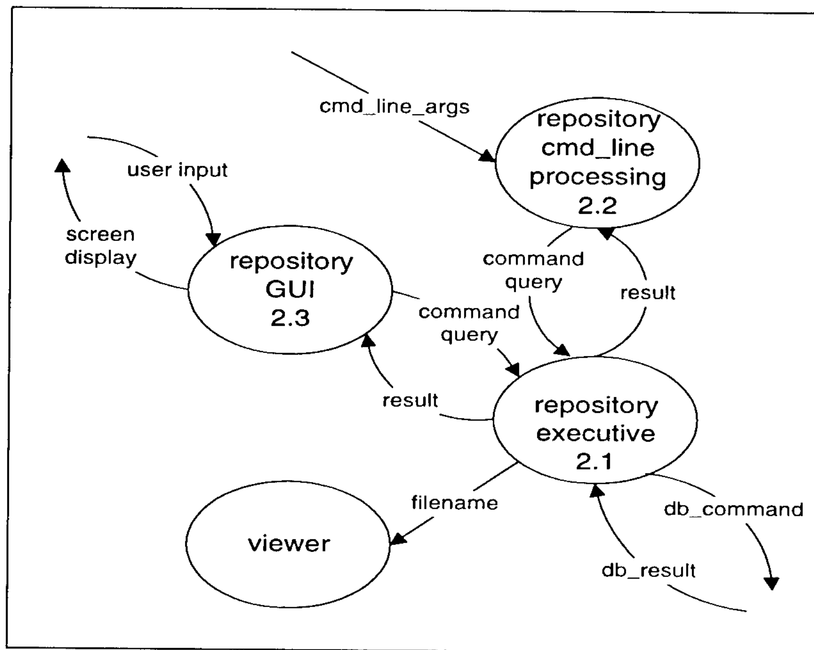
## User Interface Design and Implementation

The RABS user interface consists of a text mode windowed interface featuring pull-down menus and multiple overlapping windows. This interface is based on an existing textwins module elaborated on by the team working on the Text-window User Interface (TUI) for the repository manager and the build manager during the Fall 1996 semester.

Both the repository manager and the build manager can also be accessed via an extensive array of command line arguments. All the functionality of RABS can be accessed via either method. In addition, RABS accepts the use of manifest files that can further automate the use of the command line.

# Manager Design and Implementation

The RABS Manager component consists of two parts, Repository Manager and Build Manager. The Repository Manager provides users with an interface into the repository database; users can run RABS with command line arguments or with the graphical user interface to access and maintain a database. The *repository executive* of the Repository Manager provides high-level application oriented services for repository command line processing and repository GUI to manage the repository database.




Repository Manager Data Flow Diagram

The repository executive module performs the following processing activities:

- accepts exec_commands from its client
- translates exec_commands into db_commands and sends them to the repository database server
- receives db_results from repository database server
- translates db_results into exec_results and sends them to the client
- reports error messages when an error occurs in the database.

The inputs to the repository manager are:

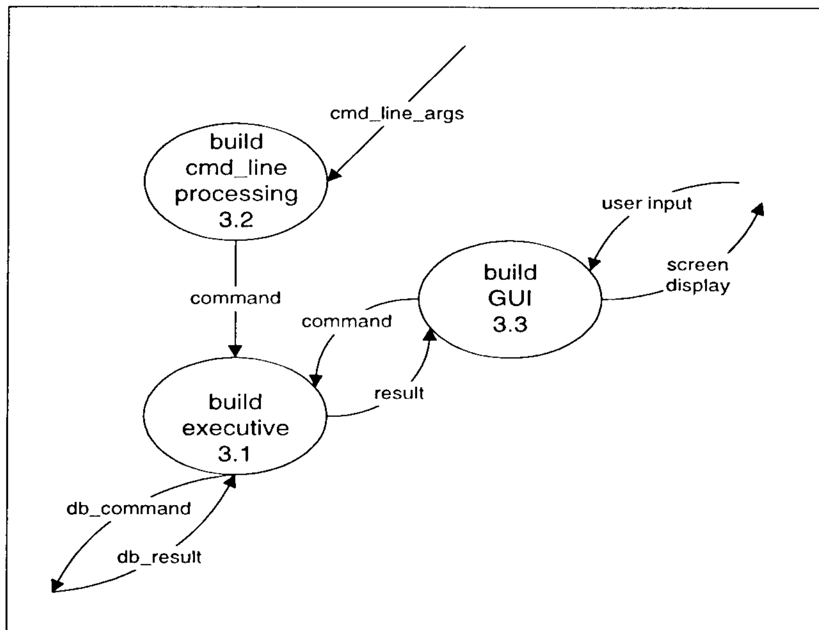
- exec_command: a request from the command line processing or the GUI (it can be any kind of database access request)
- db_results: a result from the database server, which contains an error code and/or references to a file or component objects

The outputs are:

- db_command: request to the database server
- exec_results: the result for the client, which contains an error code and/or references to file or component objects

There are two other modules, a repository cmd_line processing module and repository GUI module. The first one extracts tokens from the command line, interprets them as commands and their arguments, and sends the commands with the arguments to the repository executive to execute them. The repository GUI provides a friendly user interface to manage repository databases. It accepts users' keyboard and mouse input, controls all activities, sends commands to the repository executive, and displays results to user's terminal screen.

The Build Manager provides users with an interface to extract software components and files from the repository database. Users can either make access the build manager with command line arguments or the graphical user interface.




Build Manager Data Flow Diagram

The *Build Executive* provides build services for build command line processing and a build GUI to build an application from a repository database. The processing activities of the Build Executive are:

- accept build commands from the client
- translate build_commands into db_commands and send them to the repository database server
- receive db_results from the repository database server
- translate db_results into build_results and send them to the client
- report error messages when an error occurs in the database

The inputs to the Build Manager are:

- build_command: a request from the command line processing or the GUI
- db_result: the result from the database server, which contains an error code and/or reference to file or component objects.

The outputs of the Build Manager are:

- db_command: a request to the database server
- build_result: the result to the client, which contains an error code and/or reference to a file or component objects.

Similar to the Repository Manager, there are two modules that provide the command line and GUI inputs to the Build Manager as well.

## 8.5 Use and Evaluation of ADM tools

We intended to use the ADM model to support maintenance activities on RABS by building a plan using MS-project, devising a strategy for maintenance, arguing through requirements and design issues and tactics using RASE discussions and hypertext documents, and generating physical code structure using graphical descriptions of code relationships with ALE.

### 8.5.1 Our Evaluation Strategy

The requirements of the RABS maintenance activity involve identifying the cause of, and designing, implementing and testing fixes for a total of nine latent errors identified to be serious problems for RABS. Our strategy was to follow the ADM development process, using each of the ADM tools according to its design intent to conduct this maintenance activity. This involved the following steps:

1. create a project plan
2. export the plan from the NT server into the KBSA environment
3. create inputs required by the plan, e.g. hyperdoc list of latent errors
4. conduct analysis meetings to completely understand the RABS design and potential sources for observed errors. Capture results and conclusions using the REMAP facilities.
5. each member of the evaluation team (three active evaluators) work to develop resolutions for each of their deliverable products, e.g., identify source of latent error, develop a potential fix, discuss with team, implement using ALE, test, and finally document the resolution using hyperdoc.

### 8.5.2 Project Plan Development and Import (exercises SAM and IPSE)

We explored the SAM and IPSE models of the KBSA/ADM to a large extent. The start of any project on the UNIX platform requires that you first import the project work plan into the KBSA database on the UNIX platform (and thus use IPSE environment). This process is a routine task performed whenever a new project is created for the ADM.

The KBSA/ADM on the UNIX platform imports the project work plan from Microsoft Project, operating on the NT server, with the help of bridge software written by the Andersen Consulting team. Once the project plan is imported, the root ADM user (Super) creates users (resources) corresponding to the resources assigned to tasks in Microsoft Project. When this task is accomplished, the team members working on the project are capable of logging in to the ADM, creating sessions, and adding their tasks. From there they go on to complete the project.

#### 8.5.2.1 SAM

**Basic purposes of the tool and its advantages**

- Support other tools in providing the tool-to-tool communication.
- Provide session management features, e.g. managing views, multiple sessions, and allocation of task to session.

The main objective should be to provide seamless support to the end user in conducting work smoothly via the session-view paradigm. SAM should hide the fact that there are actually three separate tools interacting with each other to get a project done -- SAM should empower the end-user with a concept of a KBSA environment rather than a set of tools. It should be intuitive and easy for the end-user to understand their requirements (such as tasks, their composition and execution) in terms of a software development methodology such as views rather than tools.

The next important goal is to be able to restore the session context when the user logs out and logs back again. This will help in avoiding the initial setup time required to have the SAM environment ready with all the relevant topics checked out and available to the end-user without any redundant actions.

**SAM Status**

We have explored the current version of the SAM tool extensively and have determined that it does have the effect of binding and managing all the existing tools. The integration is so seamless that the end-user is oblivious to the fact that there are actually three separate tools.

The session-view paradigm is successful in making it transparent to the end-user which tool supports which views and so on. Also, because of our efforts developing the PART tool, we came away with a broader understanding of the capability of the ADM to support a user defined tool and its session-view

paradigm.

The idea of sessions is excellent. It captures the last state of the end-user's work environment and presents it to him in exactly the same manner when he logs back in later. We feel this was a good design decision and has helped avoid the extra steps needed to restore the environment, had this feature not been supported.

Overall, we thought that SAM was very well implemented. The only complaints we had about SAM were minor:

- The "description" in the Task details dialogue does not scroll, which might prevent the end-user from fully understanding what he/she is expected to do in the process of resolving the task
- Multiple task selection while adding the tasks to the session would speed that process.
- Inability to login as a different user without the need to shutdown KBSA and start it up again was occasionally frustrating.

In summary, SAM smoothly integrates model-view-controller activities, supports addition of new topic types, can add deliverables, and supports multiple sessions so users can organize their work by allocating their existing tasks to specific sessions. Tool registration and messaging structure make it realatively easy to add a new topic type and new tools.

#### 8.5.2.2 IPSE

**Basic Purpose of the Tool and its Advantages**

The IPSE tool is supposed to handle planning and performance activities, managing work objects, updating development history for work objects and thus can handles the following tasks:

- Planning
- Enactment
- Assessment

It should control the task in its entirety by manipulating the details of a task object and allow basic activities such as create/delete/view information/change state etc.

The IPSE should also be responsible for "task enactment", which includes the following activities:

1. Perform evaluation transformations
2. Launch tools
3. gather resource and work objects
4. Upon success, update the state/version in memory and database

Another important responsibility of IPSE is to support the tool integration strategy:

- Represent and communicate with all tools internally via tool proxy
- Support for COTS (Commercial Off the Shelf ) tool integration

The second issue is important, as the current implementation of KBSA uses MS-project, which stores the project plan and exports it to the KBSA/ADM (IPSE) which populates the database structure with an image of the current project plan.

IPSE has the means of guiding the end-user when he/she has made some obvious mistake or diverted from the guidelines for a particular project through the use of critic resolution tasks. The levels of intelligence in these suggestions are situation dependent.

**IPSE status**

The IPSE in KBSA/ADM was well exercised since the first activity required to start a project is to create a project plan in MS-project and export it to the KBSA/ADM environment using the bridge software. On the KBSA/ADM side IPSE becomes active and creates necessary objects, e.g., project, resource, task, and deliverable in the ObjectStore database and thus SAM is able to get its required information upon launching of the KBSA/ADM tool on the UNIX platform. This export is proof of the support available for a COTS tool in the KBSA/ADM environment.

Some observations regarding IPSE are:

1. We believe it needs to support more than one resource to a single task in MS-project and the corresponding export to KBSA/ADM environment. A resource is the developer identified to resolve a task, and only the resource has the right levels of access to do that resolution. Often, however, several developers need to collaborate on the resolution of a single task.
2. It needs to support deletion/modification of tasks and re-exporting them to KBSA/ADM. We understand the large development effort involved for realizing this on the KBSA/ADM side, but it is

a feature that is undeniably important for complex projects.

3. It should have the capability to edit task titles once they are in ADM (e.g a spelling mistake while defining the task in MS-Project should be correctable from within the ADM)

The IPSE task enactment is a well-implemented feature of the system. It reduces burdens on the end-user to select, checkout and open views on the related input topics and create and open new views for the deliverables. IPSE enacts the task and automatically opens all the relevant views and makes the environment ready for the end-user to start resolving the task at hand.

IPSE currently provides manifestations of an assistant in KBSA, which are called Critics. These critics are analysts which both check models for desired properties and suggest alternatives by adding a task for the user which is called a critic resolution task. IPSE Critic expertise is in the early development phase, evidenced by the fact that the ADM only supports three such critics, namely: Content critic, task completion critic, and a cohesion critic; all of which are limited in complexity.

Our vision of an ideal "critic" capability for IPSE is as follows:

- Different projects in different organizations have their own style of documentation or standards and guidelines for documentation and code development. IPSE critics should have the capability of "plugging in" a style grammar from the user when the project was created.
- The above suggestion requires some minimal support for a public interface (API) for adding, removing, substituting and verifying the style/standard module for a particular project.
- A set of templates of critic styles can be developed and allow the user to select one of them (if it suits the end-user) or allow the user to " plug in" their own using the API.

In summary, IPSE provides effective management of project structure, association of tasks with sessions, and the addition of new deliverables. It supports the model-view-controller model dynamically by sending messages to users when a checked out product changes. IPSE coordinates with the ADM's persistence mechanisms so that the user always opens the latest version of a topic. Its greatest asset is that it quietly does its business without intruding on the user's view of his/her work. Its greatest weakness is that new tasks and inputs to tasks can not be added after a project plan is exported from the NT server.

#### 8.5.3. Use of Hyperdocs and REMAP Discussions (RASE)

An attempt was made to use RASE to create and manage informal documents and specifications related to the RABS project. RASE supports two different views for its components, hyperdocs and REMAP discussions.

##### 8.5.3.1. Hyperdocs

**Basic Tool Concept and Advantages**

The motivation behind using the hyperdoc editor is to create a loosely structured document, with links that enable the user to traverse to various other project deliverables that are related to it. This is a very useful way of meaningfully linking together many tightly and loosely related project components. Software requirements documents can be linked to the discussions on the module designs and to test descriptions. Design discussions can be linked to the code specifications that follow, etc. Thus, the features expected of a tool that supports hyperdocuments are twofold: (1) An editor that supports basic document writing and (2) A mechanism for linking the hyperdocument with the various project deliverables.

**ADM Hyperdoc Status and Some Comments**

We've tested the ADM hyperdoc facilities in the RABS project by creating various kinds of documents: Bug lists, prioritized buglists, bug definition documents, bug-fix design documents, test reports, etc. We tested the object linking mechanism by linking the requirements documents to design discussions, code specifications to requirements and test documents etc.

The KBSA/ADM environment provides an excellent linking mechanism between heterogeneous objects, called *object linking.* This framework is an easily extendible one, and it provides the client with the capability to add a link between the various topics present in the ADM environment. Since the topics are versionable, there is a question of which version of the topic should the link point to. The objectlink mechanism solves this problem by automatically pointing to the latest version of the topic. The objectlink mechanism is put to good use in the ADM hyperdocument tool. We've found the "hyper" aspect of the hyperdocument tool to be based on an excellent design, and it works very nicely. The links can be made bi-directional as well. A suggestion for future work might be to make the hyperdocument compatible with the current standard for hypertext, namely HTML.

Moving to the "document" aspect of ADM hyperdocuments, The document editor itself serves the

purpose of a proof-of-concept. Many more features such as redo/undo, different font sizes, text justification and alignment etc. have to be added to make this editor accepted in a professional environment. Moreover, since people are already familiar with many commercial document editors, the keyword macros and functions have to be modeled closely on such products to make the ADM editor easily accepted among software professionals. Another suggestion for future work: To improve the usability of the tool, functionality needs to be added to the various (currently disabled) menu options such as "insert picture" etc.

In summary, Hyperdoc supports the import and export of standard text files, creation of object links, and can export selected text to a REMAP discussion or ale class diagram. Its implementation is not as advanced as commercially available tools that handle basic hypertext functionality.

#### 8.5.3.2. Discussions

**Basic Tool Concepts and Advantages**

It makes a lot of sense to include a tool that captures discussions in a project management tool. Discussions provide an excellent means for capturing design history, implementation strategies and how they were arrived at, and maintenance histories of various products associated with a deliverable. The embedded object linking mechanism makes this tool even more powerful, since it can also link to it the various other deliverable associated with the project (just like the hyperdocs).

**AMD Discussions Status and Some Comments**

We tested the ADM discussion topics by trying to use it to capture a couple of design discussions that we had at SU, (RABS overview meeting, RABS bug fix prioritizing meeting etc.), and the weekly conference calls that we had between Syracuse University team and Andersen Consulting.

The various discussion nodes provided in the editor (such as Issue, Position, etc.) are well thought out, and adequately capture the different kinds of situations in a real world discussion. The tool usage is intuitive and the object links work very well.

The current implementation of the discussion tool is based on the ADM's extended model view framework (XMVF) concept. The views on the discussions are not saved; they are generated dynamically every time the user requests it. This has the advantage that the views need not be stored in the database (minimizing the database size). But it has the disadvantage that the user cannot layout the discussion in a particular format on the screen and expect it to appear that way when a view is created on

it later. This problem can be solved to a some extent by editing the document in the manual mode, which allows one to *place* the nodes wherever needed, but the *routing* between the nodes is automatic, and messes up the document. The user must be given an option to turn off the automatic placement and routing mechanism.

To make this tool acceptable in a professional environment, it must provide some additional support in creating discussions. This could be modeled along the lines of wizards in MS-Windows. Right now, it serves as an electronic blackboard, and it is up to the user's imagination to capture the thread of discussions.

In summary, REMAP discussions can capture a discussion using the REMAP editor, and can create arbitrary links between requirements, positions, assumptions, decisions, arguments, and issues. Its model is based on a very good idea – creating navigable threads through important discussions and clearly documenting assumptions and conclusions. Our only significant dissatisfaction with REMAP is its auto-routing feature which, for complex discussions, results in link networks that are hard to read and trace.

#### 8.5.4 Specification Development (ALE)

We believe the ALE model intends to support Bertrand Meyer's "design by contract" using pre and post condition constructs[5]. It generates C++ code from graphical Object Modeling Technique (OMT) diagrams, introduced by Rumbaugh in "Object Oriented Modeling and Design". The goal of the ALE model is to directly support implementation by specification. This is very ambitious and especially difficult for a language as complex and context dependent as C++.

#### 8.5.4.1 Discussion

Our attempts at evaluating ALE have been somewhat frustrating and disappointing. Since working on RABS is a maintenance activity and the code for it already exists, the evaluation method we came up with was to choose two header files with at least one class definition apiece from one of the modules and see how closely we could re-create them using ALE. The module chosen was the viewer, mainly because it had two modules with class definitions that seemed ideal for our purposes.

On the first try, ALE would not cooperate at all. The names of the two header files had changed, and neither SAM nor ALE provided any means of altering the names once the project plan had been imported. If we wished to change them, we would have had to start over from scratch with a fresh project work plan

[5] These constructs were introduced by Meyers in his Eiffel programming language. It appears that ALE intends to extend this capability to the C++ language.

and a new database, losing a significant amount of work in the process.

Since that route was out of the question, we decided to just ignore the names of the modules and try to create them with the wrong names. We got as far as being able to create a package definition using the ALE package editor. The process is to draw a package rectangle in the edit window and give it a name, then double click on the highlighted rectangle. After doing this the class editor appears (it is very similar to the package editor in appearance). Once the class editor is running, you are supposed to create a rectangle in the edit window and when you do so, a dialog appears which then allows you to give the class a name and create its data structures. During our first attempt at this, the dialog in which to name the class would not work at all. The only button that was not grayed out was the cancel button. Upon canceling that dialog, the rectangle drawn in the edit window disappears. Try as we might, we could find no way to create a class.

Several days later we made a second attempt at it. This time, we were able to successfully create a class. The ADM seemed to be working better that day (possibly because only one client was running at the time, whereas the previous attempt we had both workstations running the ADM concurrently). This second attempt resulted in a successful code generation. However, the code we generated turned out to look nothing like the original header file for the RABS modules viewer.hpp and bufmgr.hpp. ALE made all of the functions virtual, and concatenated the package name onto the name of the class, its constructor and destructor, and all of the member functions.

A number of other things annoyed us while using ALE:

1. If you create a package or link two packages, then try to delete the package or the link, the ADM crashes
2. Many of the dialogs and menu items were non-functional, which caused a lot of confusion
3. Some of the functional menu items and dialogs in the package and class editors became non-functional at times, then work again after restarting the editor
4. The class creation dialog is not very intuitive, it took us a long time to figure out how to use it.
5. We noted many "ObjectStore Exception" and "Deadlock" error and warning messages while trying to use ALE, some of these caused the ADM to crash.

#### 8.5.4.2 Conclusions regarding ALE

Most of the problems we encountered relate to user interface bugs and not necessarily to any serious flaws in the design of ALE. Unfortunately, these bugs hindered our evaluation of the tool to the point of nearly abandoning the effort. We believe the tool concept to be sound, and the frameworks used in its creation

are good. Our suggestions for improving ALE are as follows:

- Fix the bugs noted above
- Improve the user interface for the class creation dialog
- Remove or make active all the menu options and items that aren't currently activated

The ability to create OMT diagrams is very useful for a CASE tool. Our final suggestion relative to ALE is that the ADM should definitely include a tool that provides diagrammatic capabilities. But this tool needs to be a fully-featured OMT diagrammer and be used to guide developers in creating code, but allow an option to bypass code creation.

In summary, the ALE structure supports Bertrand Meyer's "design by contract" paradigm and high level, specification grammar driven, implementation based on the use of OMT diagramming tools. ALE can create packages and classes, populate the classes with members, create inheritance and aggregation relationships, and generate code. We think this is a great idea, but the implementation was not effective. The user interface was not very intuitive, it's hard to make classes take the specific form you want because it has to be done through ARGO, and the ALE critic provides only a subset of warnings already available in C++ compilers.

## 8.6. Overall Results and Conclusions of the RABS Maintenance Activity

We were successful in some ways and unsuccessful in others[6].

+ The entire maintenance activity was a well-organized effort between four people. Everyone knew exactly what everyone else was working on and there was no overlapping of tasks.
+ We were able to capture meeting minutes using REMAP and referred back to them, proving that discussions are useful.
+ The bug-list hyperdoc was helpful. Each time a new bug was fixed the list was updated, giving everyone a clear picture of the project status. Other hyperdocs captured resolutions of the tasks associated with each bug.
+ The results of the maintenance activity are now clearly documented and archived within the ADM database for future reference.
+ Using the KBSA/ADM helped us organize our work in stark detail. The process we followed is reasonably well documented as a result of our adherence to the project plan.
– Tasks such as actually fixing a bug, which involved no interaction with the ADM to resolve, found us drifting away from the methodology that we had laid down.
– We often found ourselves working on source code on the NT machine then moving to the UNIX

[6] Positive results are indicated by plus symbols and difficulties by negative symbols.

workstation to use the ADM to create our deliverables. A windows NT based version of the ADM would remove a lot of the clumsiness we encountered moving between platforms to get our work done.

+ We think object linking is a very good idea. Any reasonably complex software development has many quite complex dependencies between deliverables, e.g., specifications, plans, and reports for requirements and design and implementation and test. Use of an object repository and the ability to make visual and navigable links between the objects stored there could be invaluable in organizing, tracking, and controlling developing software baselines. The implementation of object linking in the ADM is not perfect, but good enough to demonstrate how valuable that capability can be.

– We felt the ADM needed a stronger role model than SUPER versus all others. Roles would allow deliverables to be modified by some individuals and not others based on their role type, e.g. architect, team leader, team member.

– The Specification against intent strategy did not work well. Use of ALE was minimal. We had trouble due to bugs in ALE. Also, it did not fit as well with maintenance activities since the code already existed.

+ We believe the Specification against intent idea is appropriate and useful and could work well with refinement of the ADM. We use a similar (but weaker) strategy in our software design classes at Syracuse University to generate modules with documentation pages based on a module construction grammar. It would also be very useful to invert that strategy and use C++ as an architectural description language – in a sense making the software generate its own high level documentation by creating OMT diagrams directly from its source code.

## 9. Final Conclusions

There is a lot to like about the Knowledge-Based Software Assistant Advanced Development Model. We favor its goals, like the architecture and implementation of some of its tools, and believe it provides proof of concept for some very useful ideas. There were specific implementation details that did not work well along with many that worked very well. There are parts of the ADM implementation that use interesting new and still evolving technologies like CORBA and other parts that use technologies like hypertext that are now better implemented in commercial products.

In Section 7. we summarized our conclusions from the PART study, and in Section 8. we summarized conclusions drawn from the RABS maintenance activity. In the next few pages we draw together over-all conclusions about the ADM's technology demonstrations and make suggestions for extensions.

### *ADM Architecture*

The ADM architecture seems to us to be ideal for this kind of application. It is well organized, flexible and extendible. Its tool federation concept is an important feature of the architecture and works well, as demonstrated by our ability to add a new tool that extends the ADM's model of the software development process. Furthermore, we like its component structure. The inclusion of an object-oriented database seems especially important, allowing the capture of complex relationships and information about an evolving software project. Use of a session manager (SAM) to implement the model-view-controller paradigm and coordination through integrated process support (IPSE) to manage task enactment is a major success for the ADM. It supports the process model advocated by the ADM without intruding on the users' focus on their activities.

We like the concept of critics and believe a strong implementation would be very useful for assisting the software development process. Development under detailed process specifications like DoD-2167A requires a lot of cultural and technical knowledge that can be aided by the use of well thought out critics. Critics could also be very useful to support the reuse of large complex class frameworks which require a lot of knowledge about their Application Program Interfaces (APIs) and about the details of their implementing languages. The ADM's implementation of critics is a step in that direction but needs a lot of work to be useful in the senses described here.

The ADM's client server architecture uses a fat UNIX server to house the repository and very thin UNIX clients to support collaborative multi-user work flow. We observed a lot of network traffic when a user logs into one of the thin clients, and suspect that a more even distribution of processing would benefit ADM performance. We see this as an implementation issue for production equipment, not a specific

criticism of the ADM. The architecture clearly supports collaborative work in a team-based environment through task control and version control.

### Object Linking

The object linking mechanism is a very important feature of the ADM and its value has been demonstrated in the activities carried out in this evaluation. Software development for even modestly complex projects involve a very large number of issues, decisions, concepts, and products. Establishing associations among them and being able to trace the associations seems to us to be critically important, and lacking in any development environments we have used before.

The ADM implementation is not particularly robust. We have often crashed the ADM complex while navigating links. Furthermore the use of multiple links to bring up simultaneous views into specific products did not work effectively in an earlier version, crashing frequently, and seems to have been replaced by a serialized sequence which works but is clumsier to use[7].

### ADM Tools

We discussed the ADM tools SAM, IPSE, and RASE extensively in Section 8. Please refer to the conclusions of that section to find our evaluations of these components.

### Frameworks

Implementation of the ADM functionality appears[8] to be based on a CORBA backbone tying together commercial tools:

| | |
|---|---|
| ObjectStore | - object oriented database repository |
| Galaxy | - graphical user interface framework |
| MS Project | - project planner accessed through bridge from CORBA to OLE |

These tools are augmented with an extensive set of C++ frameworks used to translate the tools' functionality into specific ADM activities.

---

[7] These observations are based on a very brief look at the latest version, installed 15 days before the end of our evaluation contract, so we don't have a reliable picture of this functionality.

[8] We have only glimpses into the internal software structure necessary for our PART tool addition. We enjoyed extensive support from Andersen for this activity and so our knowledge of the frameworks is very incomplete.

ADM frameworks offer:

1. Persistent topic creation and management support, check-in and check-out facilities with version control.
2. View creation and management support.
3. Object linking support.
4. Tool to tool communication
5. Multiple session and multiple user handling

Galaxy is supported by the frameworks GEF, AGL, and Scibbles that extend and augment Galaxy to provide graphical representations of OMT diagrams and object links. XMVF provides content security and audit trails for all products managed by the ADM. PVM provides check-in, check-out functionality and manages version control, using the mechanisms inherent in ObjectStore. RO hides socket level programming and defines a message structure for the ADM. Other frameworks provide functionality for the tools SAM, IPSE, and RASE.

We were impressed with the utility and functionality of the Andersen developed frameworks. Our experiences introducing the PART tool, described in Section 7., convinced us of the value of these components. Our one outstanding criticism is that there is virtually no documentation to support design activities using them. This means that a potential adopter of the ADM technology would have to reinvent them or do a lot of reverse engineering to use them successfully.

## Extensions

We believe there are interesting opportunities for extension of the ADM facilities that could be very useful in a production software development environment. These suggestions are based on both our evaluation experiences with the ADM and from our own professional experience doing software development under DoD-2167A, and standards imposed by a variety of NATO contracts[9]

1. Support use of multiple resources for a single task.
2. Provide a flexible, role-based planning mechanism, perhaps using a complete implementation of the PART tool that supports the notion of Project Manager, Architect, Team Leader, and Team Member, each having their own accessibility rules.
3. Provide support for a template-based user defined critic mechanism in addition to more useful default

[9] The Principle Investigator worked as a software developer, system engineer, and software manager for many years, including work on a system involving several million lines of source code, created by a team of hundreds of developers on both coasts of the United States. The system has been fully operational and met its functional and performance requirements.

critics.

4. Provide syntactical support for DoD-2167A DIDs and project defined style guides. A grammar-driven assembly of documentation, supported by wizards with specific DID knowledge could be extremely useful.
5. Support development of qualification tests by grammar-driven analysis of requirements captured in a structured database based on the ADM's repository.
6. Provide the capability to baseline products on a project wide basis for off-line storage, purging intermediate versions from the object database.
7. Allow versioning on a selected basis determined by the "Super User".
8. Incorporating signature analysis, as Dr. S.K.Chin is developing with his team at Syracuse University. This could solve some thorny problems in project management and during Functional Configuration Audit (FCA) and Physical Configuration Audit (PCA). Signature analysis could provide the control necessary for authorizing requirements changes, approval of formal reviews and resolutions of action items, and changes to a formal baseline.
9. Directly support software reuse by incorporating a RABS like repository structure in the ADM.

# Appendix A: PART Design Document

## Functional Model for PART

The ADM environment currently has tools for generating project design documentation such as Hyperdocs and REMAP discussions. It also has a facility for specifying code components, resulting in the generation of header files. A real-world software development process definitely needs the above project planning, and topic-development support. In addition, it also needs some facilities for managing the entire software development process, such as:

- Support for module and testplan topic types
- Support for managing and browsing software products
- Support for spawning documents: manuals, testplans, reviews etc.
- Support for the Software Integration process: code builds

This motivates us to introduce the Project Archival and Report Tool (PART).

PART introduces the concept of a **Project Knowledge Structure** (PKS) in an attempt to satisfy the above requirements. A PKS is essentially the MS-project plan, augmented with some additional information. Hence, it contains information such as the list of tasks, their inter-dependencies, the resources allocated to the tasks, their deliverables etc. A PKS is shown in Figure 1. Each box in that figure corresponds to the notion of a task in the ADM world. Hence, the tree represents the inter-dependencies of the tasks between themselves. PART will display a graphical representation of the PKS in its window, The PKS can be shown either for the entire project (global view), or for a single session within the project (distributed view). This is discussed in more detail in a following section.

What's the idea behind showing the PKS to the user? The intention is to provide a graphical interface to the project itself - the user can browse a project's tasks and deliverables in an easy and convenient manner. PART also supports **visitors** and **builders** towards this end; visitors allow the user to browse the PKS according to some search pattern, where as the builders allow the users to generate documents and build code.

Visitors present different kinds of views to different kinds of users; for example, a *general's tour visitor* might be targeted towards someone in the higher management, who wants to see only the broad divisions of the project and the major design issues, without getting into the details. A *team leader visitor* might be designed such that it presents a lot of details about a particular process within a project..

A **build** refers to a process by which deliverables are selectively pulled out of a repository to form a meaningful collection. With this facility, developers will be able to generate execution images for modules, manual pages for the sub-modules for a process, test plans for a multi-module component etc.

PART will need to add a couple of new topic types to support these goals. Specifically, PART will support a **module topic type** and a **testplan topic type**, and the associated views for them. A module topic represents a software module with a header file and an implementation file, and a testplan topic represents a software testplan.

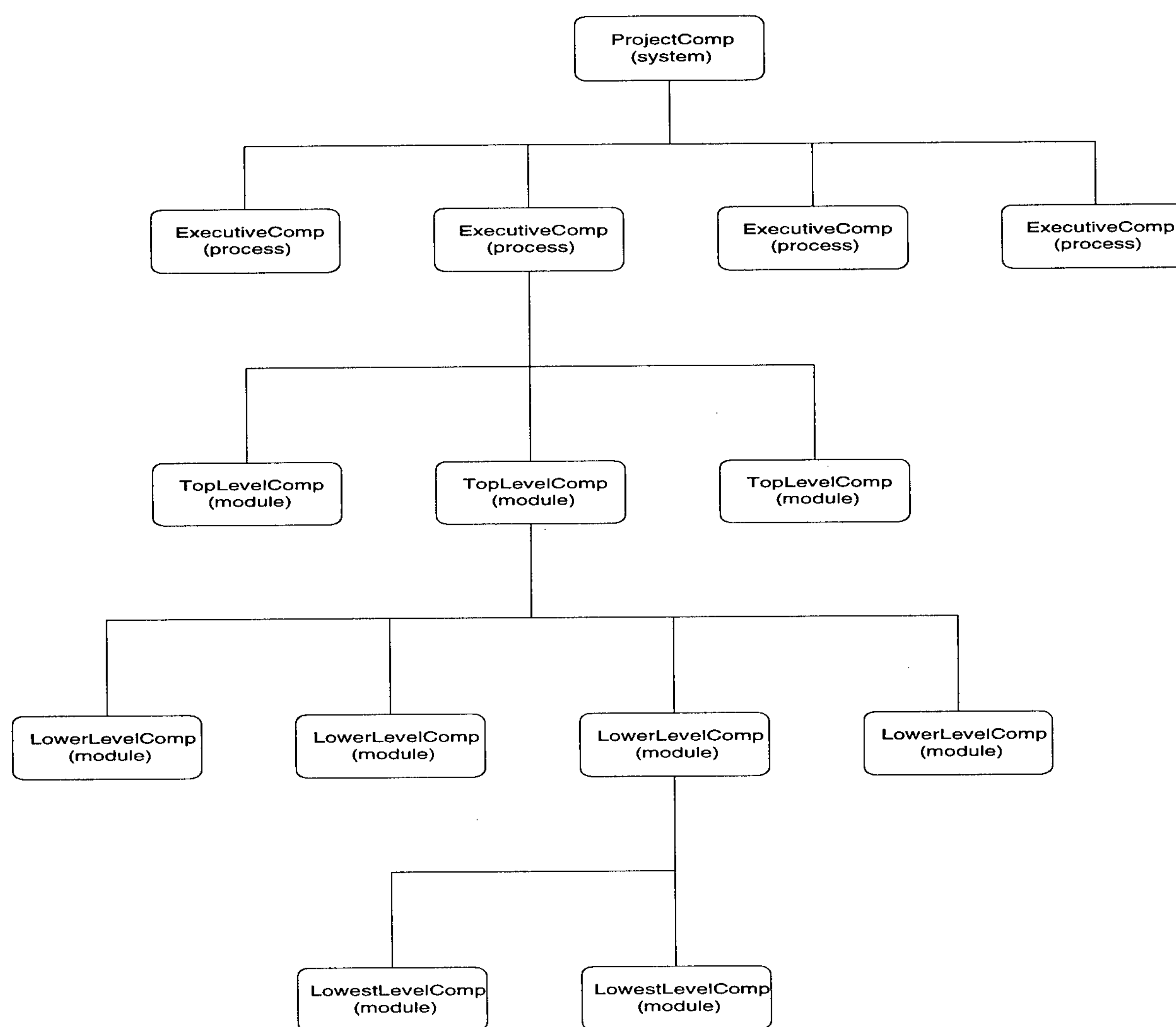



**Figure 1: Project Knowledge Structure**

## Conceptual Model for PART

The aim of the conceptual model is to illustrate the relation between the ADM's notions of tasks, deliverables etc. to PART's notion of components, products etc.

The following diagram shows the connection between the ADM world and the PART world.

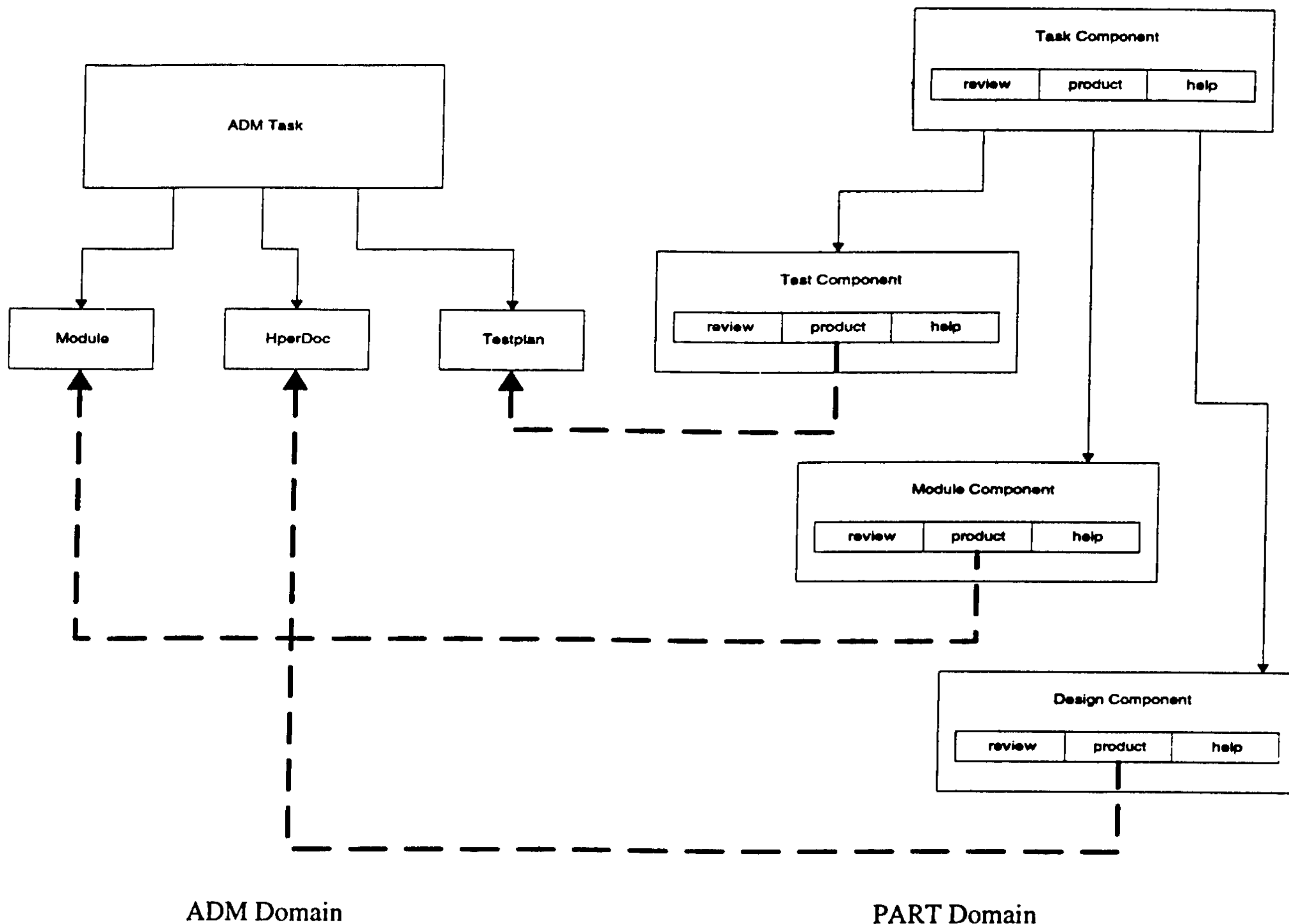



Figure 2: Conceptual Model

The dotted lines in the above figure show the runtime pointers made from the component structure to the actual topics created by the ADM.

During our design efforts we tried to focus in on what functionality PART should encompass and what its boundaries are. Some of the assumptions made are:

- PART is able to traverse the whole database file
- The Module topic always consists of one header file (.hpp) and one implementation file (.cpp),

- PART does not support any kind of role mechanism (e.g. the hierarchy of Project leader, team leaders, developers)
- A Testplan topic is associated with a single module etc.
- The component structure is based on the notion that a user will want to check out the topics that are created or modified by a task. So, update deliverables and output deliverables are considered to be equivalent for the purposes of the component structure. Input deliverables are considered only for building up the task dependency hierarchy; other than that, the component structure does not deal with input deliverables.

**PART Basic Types**

**Introduction to Module and Testplan Topics**

The Module and TestPlan topics are two new topics that are designed to store, as part of the ADM database for a project, the source code and the unit test plan for a software component product. The Module topic consists of a header file and an implementation file (.hpp and .cpp files). The view for a module topic allows the developer to specify the names of the two files, and examine both files concurrently. The current interface is read only--if the developer intends to do work on these files they must be extracted from the database (by saving them to a directory) and then edited outside of the ADM environment. When the developer finishes editing the files, he then opens them in the ADM module view dialog and saves them in the database.

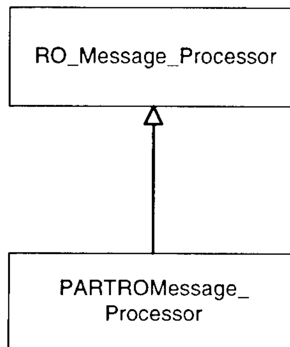
The TestPlan topic dialog allows the user to enter information relating to the testing of a component. This information is then saved in the database and can be referenced or updated whenever necessary.

Refer to section 7.3.2 for screen shots of the dialogs for these topics.

## Object Model for the new topic types

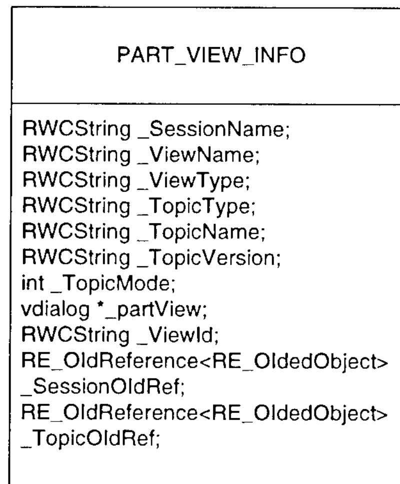
The following OMT diagrams give the details of how the various classes in PART are related to each other.

Like any other tool, PART's MessageProcessor is derived from the RO_Message_Processor.



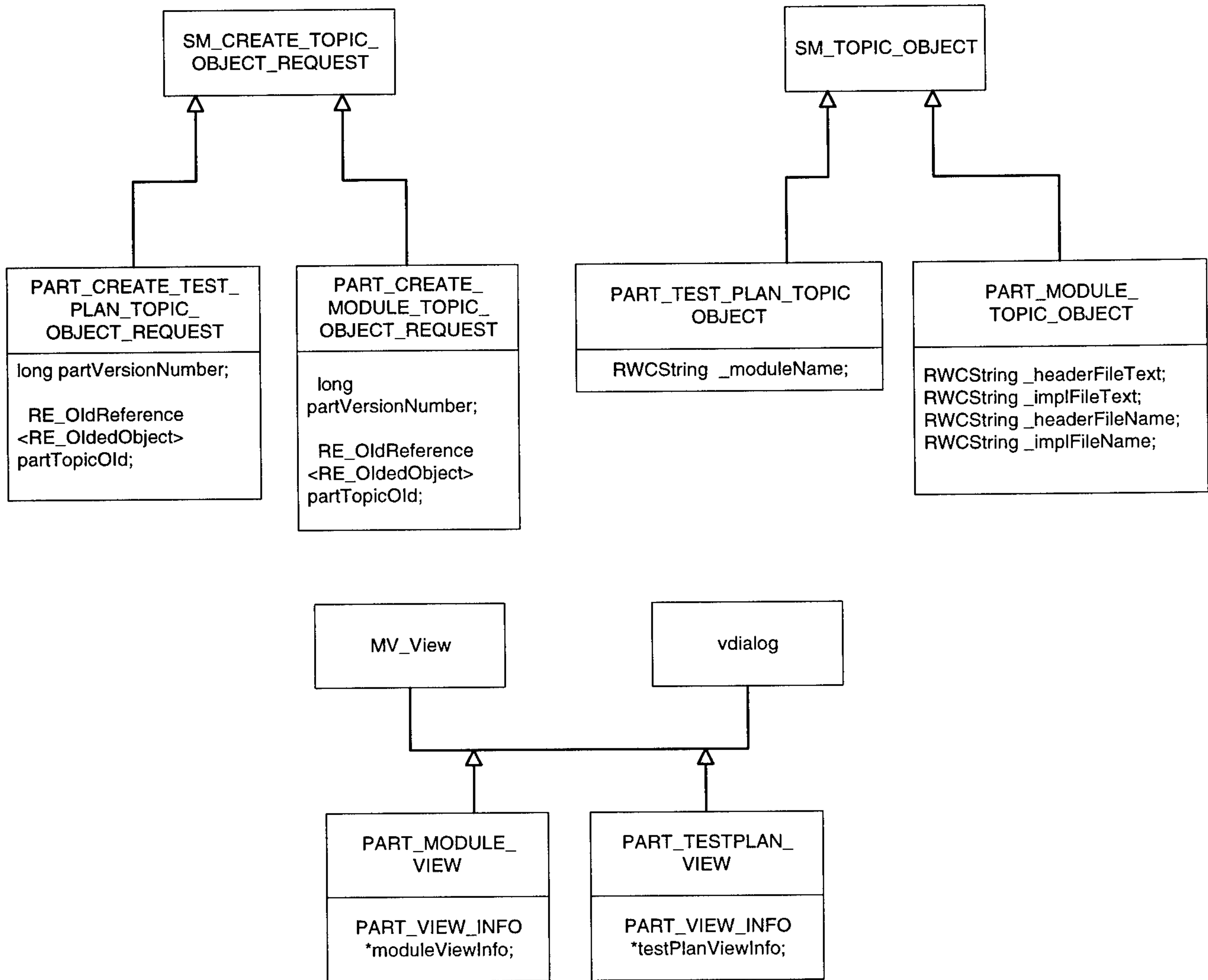

**PART RO Message Processing**

The following diagram shows the structure of the PART_VIEW_INFO class, which contains the information about a PART view. Namely the object creation request classes, and the topic classes themselves.

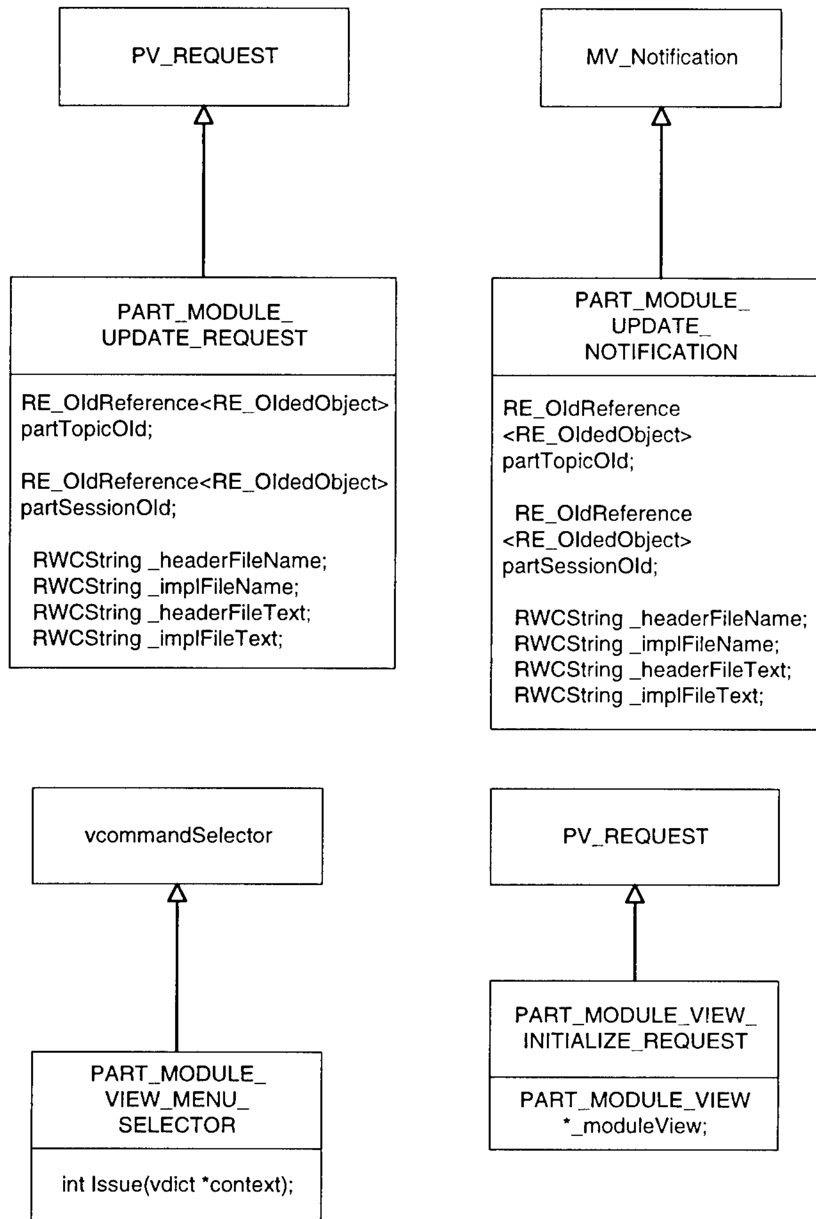



**PART_VIEW_INFO Class**

The following diagrams show the classes required to create new TestPlan and Module topic objects. Namely the object creations request classes, and the topics themselves.




**PART Module and TestPlan Topic OMT Diagrams**




## Supporting objects for the PART Module Topic

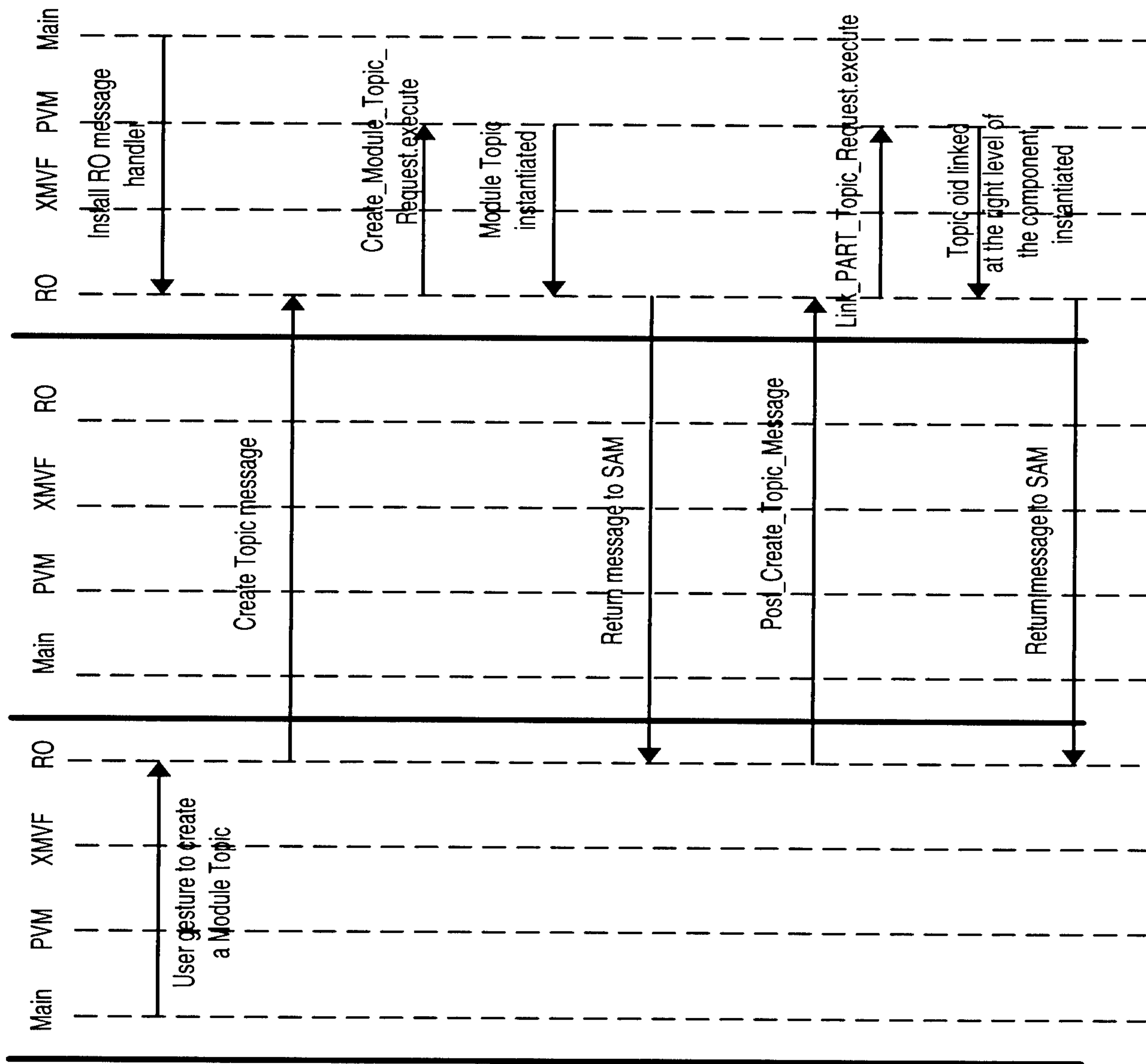
The MENU_SELECTOR class above is for galaxy event handling for the Module topic's view, and the UPDATE request is used to change the information in the persistent Module topic. Whenever a change is made to persistent Module topic, a NOTIFICATION object is used to send that information to any view that might be open at that time.

Dynamic Model for PART's topic creation

The function selected for the following event trace diagram *is Create Module Topic.* The Component structure described in the Object model will be instantiated and the project knowledge structure skeleton will be formed just after the project plan gets imported into KBSA. Now when a user in the process of enacting a task assigned to him/her creates a Module Topic or Test Topic, the following events will take place and the relevant frameworks will respond and carry out the desired functionality.

Events take place in the following sequence

- User selects the Topic->Create->Module menu sequence.
- SAM RO will send a CreateTopicmessage to the PART tool since this topic will have been registered with the SAM when PART gets instantiated.
- PART RO message handler will be active after PART main goes into its event Loop, and will receive the message.
- Message Handler for *CreateTopicMessage* will determine the exact topic type amongst the topics that PART supports(Test, Module) and generate a PVM request Create_Module_Topic_Request or Create_Test_Topic_Request as per the topic type
- PVM request gets executed at PVM layer which connects to Objectstore and creates a persistent Module Topic
- After successful execution of the request , a return message is created which is sent back to SAM RO
- SAM sends a *PostCreateTopicmessage* to the TOOL RO
- RO message handler will create Link_PART_topic_Request and executes it, which goes to the PVM layer and is responsible for searching through the component structure for the Topic Name/Topic Type combination and on finding the placeholder for the particular topic and attach the object id of the topic type which is available from the request object to the product-oid data member for the searched Component



SAM
IPSE
PART
Main
PVM
XMVF
RO
User gesture to create a Module Topic
Create Topic message
Install RO message handler
Create_Module_Topic_Request.execute
Module Topic instantiated
Return message to SAM
Post_Create_Topic_Message
Link_PART_Topic_Request.execute
Topic oid linked at the right level of the component instantiated
Return message to SAM

## Screen Shots

PART Test Plan View

Test Plan Type:

Date:

Module Name:

Module Developer:

Test Plan Developer:

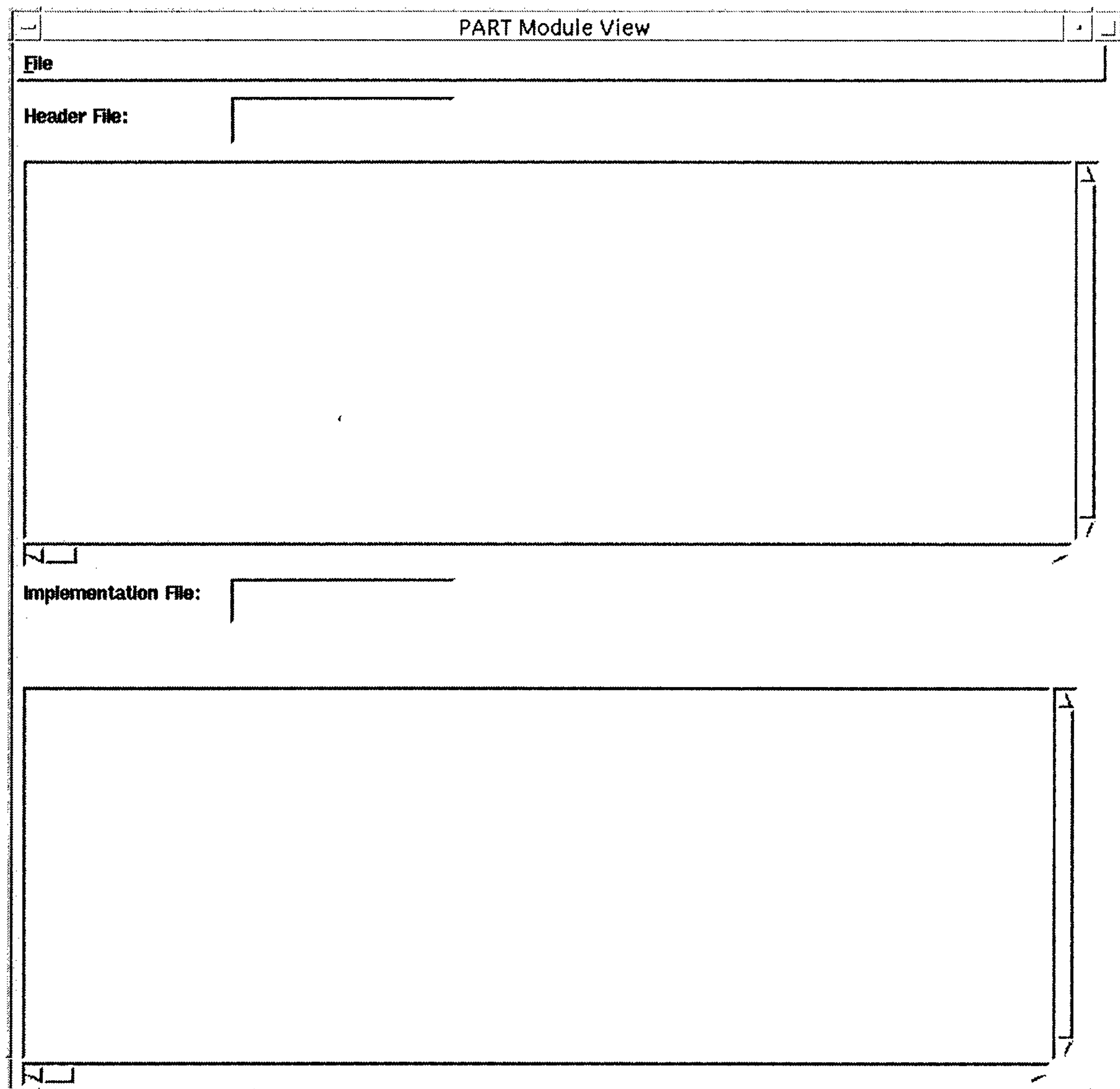
Test Case:

Expected Results:

Actual Result:

Test Data Source:

View for the Test Plan Topic




View for PART's Module Topic

## Implementation Strategy

The strategy we have followed towards implementing PART is as follows:

- Discussions to clarify our notions of a testplan and module topic
- Formulated the views for the new topics
- Wrote code for tool Registration, Unregistration, Suicide

- Wrote code to create a new testplan topic from SM_TOPIC_TYPE with no data members in it
- Wrote code to handle messages for
    - Create
    - Delete
    - Checkin
    - Checkout

- Used Galaxy to create a view for the testplan topic
- Wrote code to display the view. At this stage, the view doesn't depend on what is in the testplan topic; it is a static view.
- Used Galaxy to design a view for the Module topic
- Wrote the code to provide all the basic functionality for the module topic also (create,checkin,checkout,delete,display a different static view)
- Wrote code to make the view dependent on the topic data itself
    - Created a relation between the view class and viewinfo list
    - Created request PART_MODULE_UPDATE_REQUEST
    - Defined doIt() for the request
    - Defined a notification PART_MODULE_UPDATE_NTF
    - Identified the event loops for GUI, understood how the menu gesture is trapped as an event
    - Defined the viewers execute() loop, where the notification arrives

Advantages of using the ADM environment for tool development

The ADM environment provides the tool developer with many convenient features, which are otherwise rather time-consuming to develop. Some of these features that the developer `gets for free' are

(1) Topic Level Features: Persistent Topic Creation and Management support, Check-In and Check-Out facilities with Version control, View creation and management support, Object Linking etc.

(2) Tool-to-Tool Communication

(3) Multiple Session handling capabilities

(4) Multiple User handling capabilities

These features can be easily incorporated into the new tool with minimum effort by making use of the various ADM frameworks.

## PART Project Knowledge Structure

### Introduction

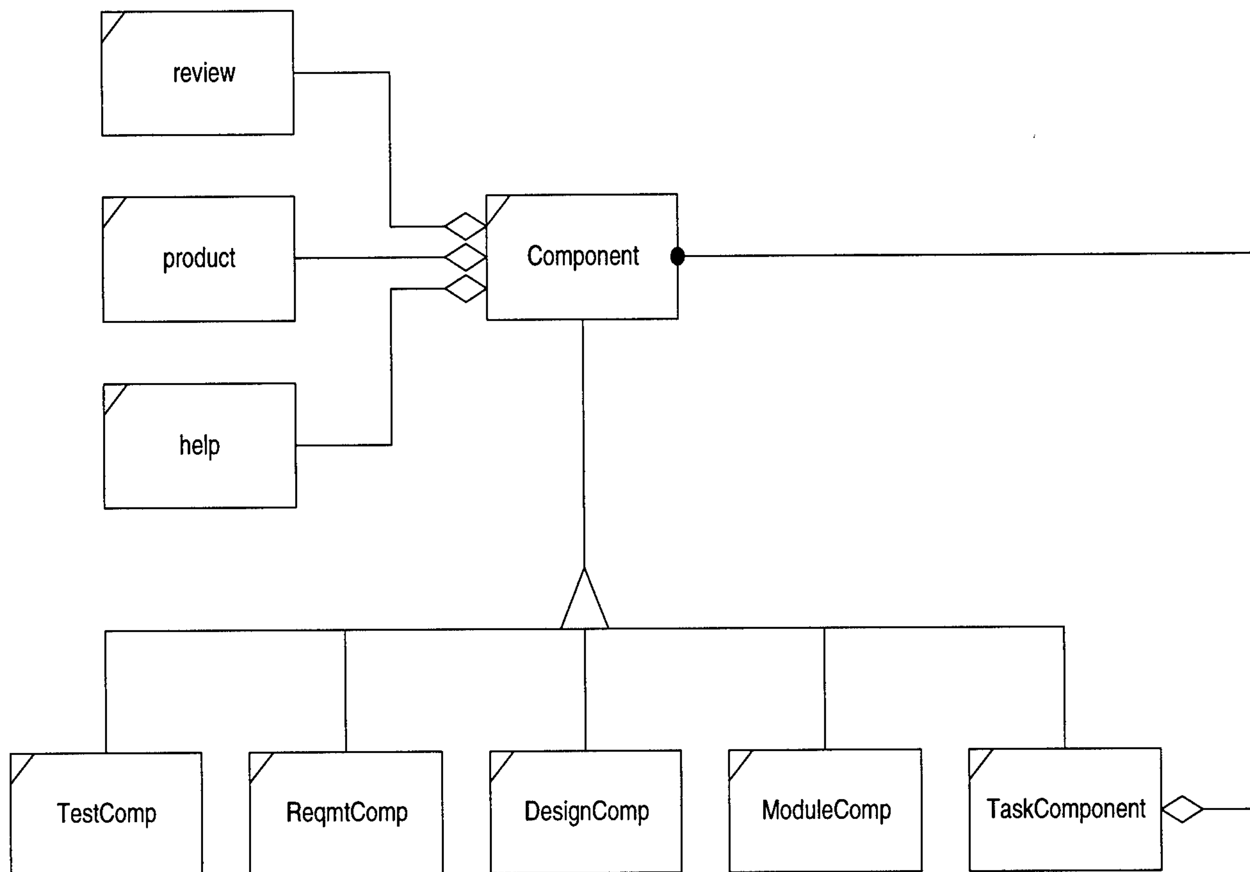
The PART dialog presents the user with a graphical representation of the task hierarchy, and the status of each of the task - how many of the deliverables have been created already, how many are being currently modified etc. PART also invokes the views for these various topics, and thus serves as an object browser for the project tasks. In this sense, the PART dialog is not a *view* according to the model-view framework. There is no *model* behind the PART dialog; all it does is represent the collection of information in a graphical way.

### Object Model

### Component Structure

The OMT diagram on the next page specifies the classes involved and their relationships for implementing the project knowledge structure. The *Task Component* is an aggregation for the whole Project Knowledge Structure. This maintains a list of pointers to objects of type Component, each of which can be either a *TestComp*, *ReqmtComp*, *DesignComp*, *ModuleComp* or a *TaskComponent* itself, which gives the structure ability to add an additional level to the Project Knowledge Structure. Thus *TestComp*, *ReqmtComp*, *DesignComp*, and *ModuleComp* are the leaf nodes of the Project Knowledge Structure.

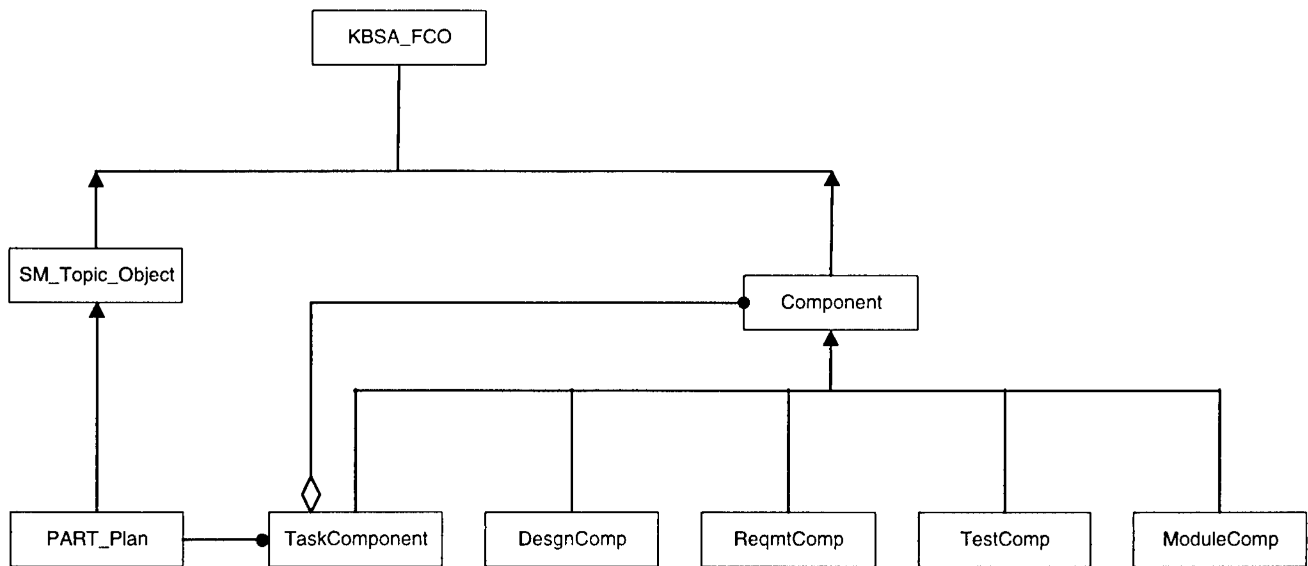
Each Component owns three objects, which are *review*, *product* and *help*. The *product* object contains the object identifier *productOId* to the concerned persistent topic object. The review class is to contain any review comments for the topic under *product* and the *help* class contains the help text for the same topic. The *help* and *review* objects can be instantiations of the RWCString class.




**Persistent Component Structure**

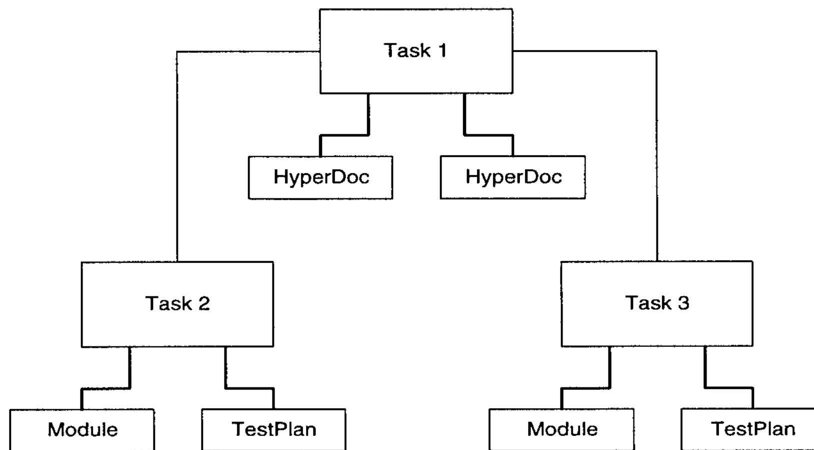
The structure described above takes care of the recursive nature of the PKS. We require the PKS to be persistent and versionable, since it is going to contain references to persistent topic object from the KBSA framework. We propose here the integration of this Component structure with the KBSA PVM layer.

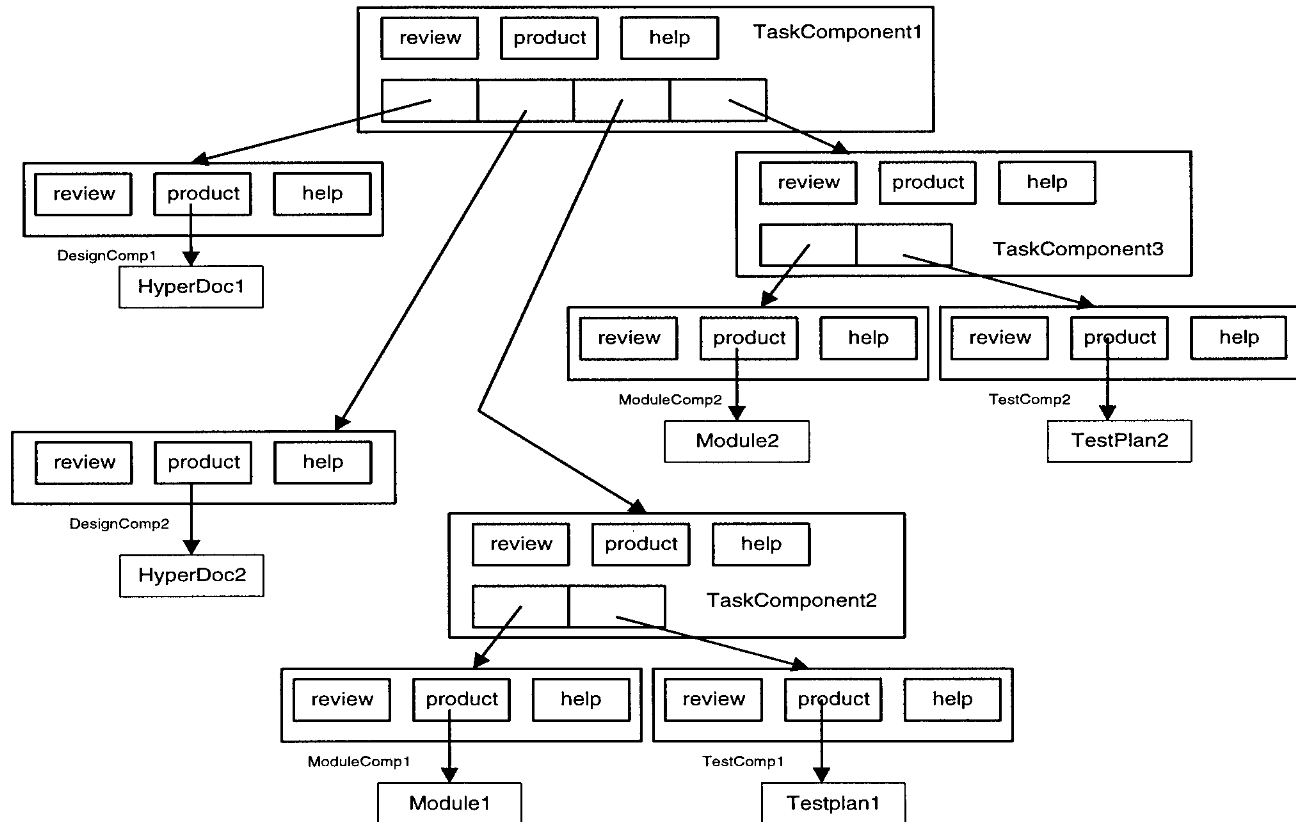
To make the component structure persistent we have introduced the Part_Plan class which is analogous to the project_plan class that IPSE makes persistent. The Part_Plan is a SM_Topic_Object, since this will make the PART_Plan persistent, as well as versionable. The Component structure classes will also become persistent but they are not needed to be versionable. This is the reason that the Component class is derived from the KBSA_FCO class; to make it persistent but not versionable. The PART_Plan class will have reference to the TaskComponent object which can be more than one, in case the PKS has more than one root node. Each such Task Component class will then have its own PKS tree.




## Object Instantiations

The object instantiations for creating the component structure inside PART is shown on the next page.




One important question to be addressed is question *when does PART attach the new topics to component structure*. This depends on whether the project view presented is a global one, or a distributed one. Here is a comparison of these two approaches:

- *Distributed Approach*: The information regarding a project is always updated and is maintained at the session level, and not the project level. The component plan is updated each time a new topic is created; PART doesn't wait until it is checked in to the database. The background activities will increase and hence the performance will suffer, but at any instant of time, the latest status of the project will be available; this is irrespective of whether any topics are checked in or out in any of the active sessions. (Important messages will be CreateTopicMessage, TopicCheckIn, TopicCheckout, etc)

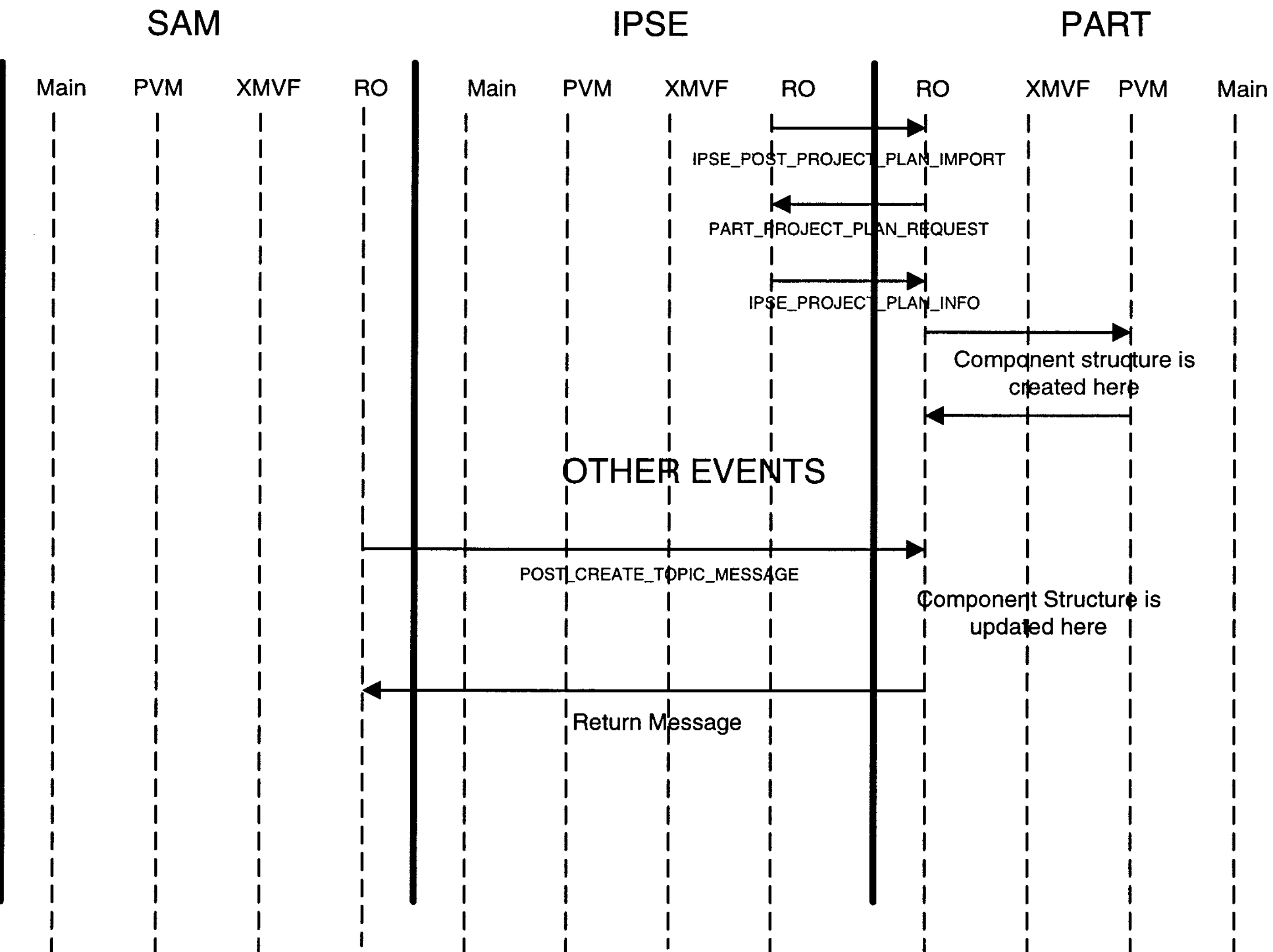
- *Centralized Approach*: The information/status is maintained at the project level, which means if there is an active session with a checked out topic, the information/status at the project level for that topic will show an earlier version of that topic. This approach improves visitor performance, since it need not bother about the active sessions. The disadvantage is that, to get the latest project status, the project manager has to ensure that all the topics have been checked in and there is no active session alive which might have checked out important topics.

1. We propose the distributed approach that is followed, then the topics need to be hooked into the component structure as they are created. So PART must respond to the POST_TOPIC_CREATE_MESSAGE, and obtain the OId for the new topic. At this instant the Component_plan is already built. When the topic_create_message or topic_checkin_message or topic_checkout_message arrives, it is required that the Component_plan be checked in to the session in the background. With the help of the topic information from the message structure the component_ structure tree is searched for the topic_name and the topic_type combined as the key to locate the placeholder for that topic in the PKS. Once the node is located the latest Object-id for that topic will be linked to the oid data member of the product data member of the Component.Thus at any time the latest topic-ois is pointed to by the component_structure. This is possible because PART already has the task hierarchy and the names/types of the associated deliverables.

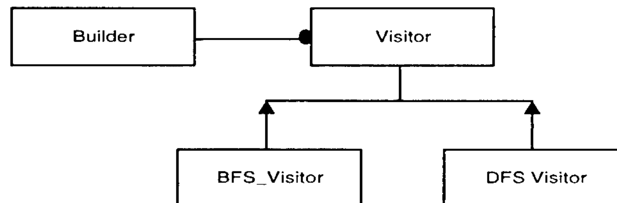
2. There are some possible inconsistencies that can occur in this situation. For example, consider the product data member of the component structure. It was set to point to a particular topic in the database, and for some reason the user deletes that topic at a later stage. What happens to the pointer now? Similarly, if the product data member was pointing to a particular version of the topic. A new version of the same topic was created at a later stage; does the pointer automatically update itself or still point to the old version. (For this situation, ADM's object linking capability can be used to make it always point to the latest version).

**Dynamic Model**

The event trace diagram in this page summarizes the above discussion and shows how the component structure is built up during run-time



## Visitors




The idea is to have the functionality of code-build and document build for the PKS. The code-build is meant for topics that are referenced by the ModuleComp objects at various levels of the PKS.

The above OMT diagram describes the design of the object model for the builder/visitor concept. We currently define two types of visitors characterized by their search pattern: Breadth First Search Visitor and Depth first search Visitor, each of which is a Visitor object. The builder has access to a visitor object, which can be instantiated and linked to the data member of the builder at run-time. The visitor functionality will be invoked with the help of menu-items in the Global PKS window view.

*Sample Build process description for Test-Plan build with a Depth first visitor strategy for the product information with destination as a file:*

- The user can specify graphically (with the help of mouse clicks, the branch(es) to be traversed from the PKS. By default the whole forest will be considered as selected. The Component structure will be traversed as follows :
- For each tree root
    - For all the TestComponent objects in the list datamember of the Task Component object, node
        - get the oid of the topic from the product data-member
        - get the relevant information stored in the topic with the help of topic object's member methods
        - Append to a temporary file
    - Visit the next node in the tree according to the classic Depth first tree traversal
- Copy the temporary file to the file the name of which was obtained from the user

## Screen Shots

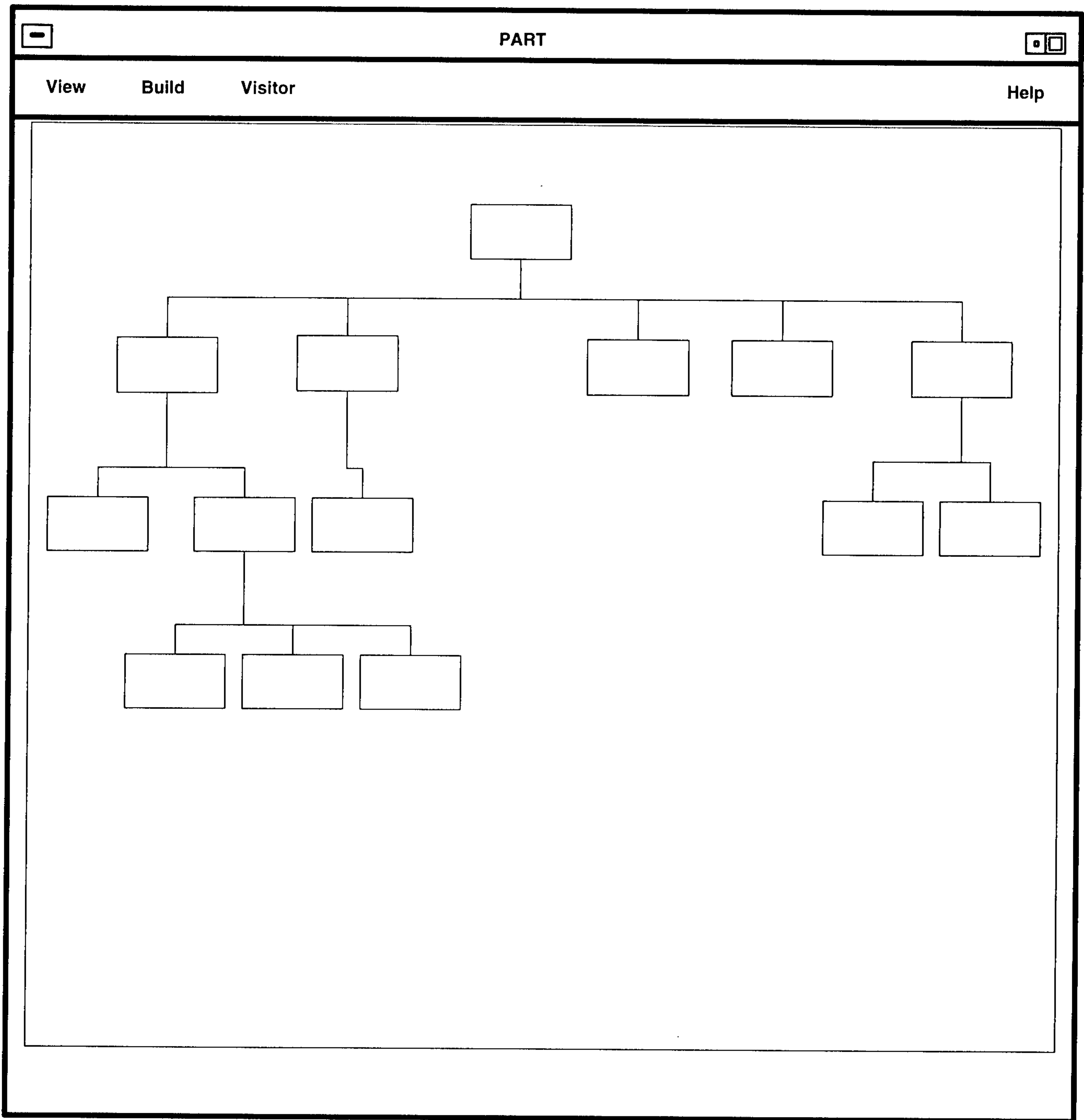
The design of the menus is shown in the following screen design: (All the sub-menuitems will be radio-buttons)

The Visit menu item will allow one selection to be made from each of the following grouping: BFS/DFS, CODE/TEST/DOCS, PRODUCT/HELP/REVIEW. Once the user has made the selection he/she can use the Build menu item to select the destination of the build, which will consist of either a temporary file generated as HyperDoc and shown on the screen or exporting the built information to a specified file the name of which can be asked to the user with the help of a File browser window.

The following dialog shows checkboxes; from BFS and DFS, only one can be selected at a time. From CODE, TEST and DOCS, only one can be selected at a time, etc.

Visit
BFS
DFS
CODE
TEST
DOCS
Product
Help
Review
Build
To Screen
To File

PART Dialog Choices




PART Main Dialog

Implementation Strategy

How is the PKS structure built up during run-time? There are two key issues to be considered here.

First of all, PART should know the hierarchy (the task dependency tree) even before any topics are checked into the database. This is because, PART might be required by the user to present a view into the project status even before any topics are checked in. There are a couple of ways in which the task dependency information can be obtained from IPSE:

(1) IPSE can broadcast a message as soon as the import from MS-Project is performed. Say, IPSE sends IPSE_POST_PROJECT_PLAN_IMPORT, a message that lets PART (and other tools) know that a new project plan has been imported. At this stage, there are two alternatives: either IPSE can broadcast the project plan information, or PART can specifically request IPSE to send the plan to it alone by sending PART_PROJET_PLAN_REQUEST.

The latter choice prevents data being sent to everyone unnecessarily. IPSE responds to the PART_PROJECT_PLAN_REQUEST by sending PART a new message called IPSE_PROJECT_PLAN_INFO.

(2) A second method avoids a lot of overhead and unnecessary transmission of data. Here, it is assumed that the PART dialog holds only session level information. So, the task dependencies for the entire project need not be transmitted. PART has to know only which tasks are present in a particular session. This information is created when the user adds tasks to a fresh session. Once the task addition process is complete, IPSE can send that information to PART in a way similar to the one described above. For the purposes of this document, this second approach is followed.

At this stage, PART contains all the relevant information regarding the project plan, such as the tasks, their dependencies, and their deliverables. PART reads task information from the message, and creates instances of the component object to reconstruct the IPSE plan in the form of a component plan; this will be only for the tasks in a particular session though. When this is created for the first time, the product data members of all the components will obviously be empty. Topics can be added to the structure as and when they are created (See 2 below). A view can be created based on the task dependency hierarchy

alone; for links to the actual topics can be provided for those that have been already created, and view should contain empty slots for all the topics that have not been checked in yet. Each time a new project plan is imported into the ADM, a new version of the component structure is created in the database.

How to start up the PART dialog?

There are a few different ways to start up the PART dialog. (1) One way is to add a new item to the SAM menu called PART, and then trap that menu selection event inside SAM's Galaxy event loop. SAM then fires up PART's main dialog inside this event loop. This requires a change (though a minor one) in the SAM source code to include this new menu item. (2) The second method is to try and keep the tools as loosely coupled as possible. One can fire up the main PART dialog as a part of the tool registration process itself. Thus, when the user starts up the KBSA environment, the PART dialog also will be popped up. Initially the window might be empty, if there is no session information available.

Implementation Priorities

The implementation of the component structure is best done in the following manner, as it leaves us in a logical state of completion at the end of each step.

1. Develop component structure based on user actions to create, check-in, and check-out topic etc.
2. Create a dialog showing the component structure
3. Enable one visitor
4. Respond to the process of adding tasks to sessions, and build the component structure based on that
5. Project-level activities and other items, such as validation issues, maintaining consistencies etc.

# Appendix B: Glossary of Terms

AGL - Aesthetic Graph Layout
One of the sub-frameworks within GEF, this API provides routing routines for the links between nodes.

ALE - Argo Language Environment
The ADM tool associated with creating packages and specifications and generating source code.

### *Argo Language*

A formal specification language for high-level software design. This language can be translated directly into source code. Argo is a more formalized version of C++. In Argo the declaration syntax is simplified, a module construct is added, and Meyer's notation of "Design by Contract" is explicitly supported.

### *Code Generation*

Translation of an Argo specification in C++ source code.

### *COTS Tool*

Commercial Off The Shelf Software Tool (i.e. Microsoft Project, ObjectStore).

Critics
A manifestation of the assistant metaphor in KBSA. Critics are analyzers which both check models for desired properties and give suggestions on how to fix the model should a desired property not be satisfied.

Deliverable
The result of a task being resolved. A task may have one or more deliverables (see *Task Resolution*)

Evolution Transformations
Editing operations that make complete semantic changes to a model. Their intent is to formalize as a single operation stereotypical editing operations which are made up of several other editing operations.

### Galaxy

Visix Software's X-Window programming API and tool set.

GEF - Graph Editor Framework
A framework created by Andersen which provides a tool set API for creating diagrams for applications like ALE and REMAP.

Hyperdocument
A text document having the ability to present the user with hyperlinks to other information. When clicked on, these links display other topics.

### Hypertext Editor

The ADM tool used for creating and viewing hyperdocuments.

### IPSE – Integrated Process Support Environment

The structure which stores the exported project plan in the KBSA/ADM environment. IPSE is the infrastructure which supports the collaborative model in the KBSA/ADM.

### KBSA/ADM – Knowledge Based Software Assistant / Advanced Development Model

The software system being evaluated.

### Methodology

A blue-print or outline of a work process. You can use the methodology to help you plan your tasks in Microsoft project. Included are task packages, or general steps to follow when creating your application.

### Microsoft Project

A software tool for managing projects.

Object Link (OL)

A logical link formed between two or more entities. For example, a hyperdocument can be linked to a position within a REMAP discussion.

### ObjectStore

The object oriented database which provides the storage mechanism for the KBSA/ADM.

### ODMESR

The Andersen Consulting Object Development Methodology, Early Support Release. ODMESR is a methodology created by Andersen in order to provide standardized practices in software design throughout the corporation.

### OODBMS – Object Oriented Database Management System

Persistent storage for objects.

### PART – Project Archival and Report Tool

The tool designed and partially implemented by the Syracuse University KBSA team with Andersen.

### PKS – Project Knowledge Structure

The information presented in the main PART dialog window.

### Project

A collection of topics that are put together for some development purpose. These topics are tasks, resources and the methodological deliverables.

### Project Work plan or Project Plan

See *Work plan*

### PVM – Persistence and Version Management

A framework developed above ObjectStore whose purpose is to define the KBSA/ADM repository and provide version management, check-in/check-out capabilities, and utilities to create databases, projects users and sessions.

### RASE – Requirements Acquisition and Support Environment

This is the ADM tool which provides both REMAP and hyperdocument facilities.

### REMAP

Discussion editor used to capture design decisions and the rationale behind the decisions.

### Resolution

The solution or completion of a task. Tasks and resolutions have a 1:1 relationship.

### Resources

People assigned to tasks in the ADM via designation in a Microsoft Project work plan.

### RO – Remote Operations

This is the CORBA implementation used in the ADM. It provides connectivity to an existing tool through ports, synchronous and asynchronous communication protocols, message based communication through abstract data types, and spawning and termination of tool processes.

### SAM – Session and Agenda Manager

This is the main dialog of KBSA/ADM.

### Scribbles

One of the sub-frameworks within GEF. Provides the capability to manage (create, control, etc.) the graphical entities such as rectangles and lines. It is an extension to the Galaxy graphical features.

### Session

A scope of the work of a user. Allows the user to organize their work in ways convenient and meaningful to them. The user only works on work objects through sessions.

## Super

The "root" user of the ADM. This user is capable of adding other users and is in charge of importing a project work plan to the ADM.

## Task

A description of work that needs to be performed.

## Task Package

A logical grouping of related tasks in a Microsoft Project work plan.

## Task Resolution

see *Resolution*

## Topic

A product of one of the KBSA/ADM tools. Original topics include: Hyperdocuments, REMAP discussions, and ALE Specifications. New topic types can be invented and added to the ADM. The SU team created topics such as Module and TestPlan for example.

## User

see *Resources*

## VRE – Visual Resource Editor

The Galaxy tool for creating X-Window dialogs.

## Work plan

The result of using Microsoft Project, a file with a .mpp extension that lists tasks, resolutions, and deliverables and links them. The finished workplan is exported to the ADM and provides the collaborative underpinnings of the system.

## XMVF – Extended Model View Framework

A modified model-view-controller framework often found in SmallTalk applications. Its objectives are to establish and maintain relationships between KBSA/ADM views and back-end object models and to provide a consistent mechanism for defining messages between the views and the model of the system.

☆U.S. GOVERNMENT PRINTING OFFICE 1998-610-130-81047

## *MISSION*
## *OF*
## *AFRL/INFORMATION DIRECTORATE (IF)*

The advancement and application of information systems science and technology for aerospace command and control and its transition to air, space, and ground systems to meet customer needs in the areas of Global Awareness, Dynamic Planning and Execution, and Global Information Exchange is the focus of this AFRL organization. The directorate's areas of investigation include a broad spectrum of information and fusion, communication, collaborative environment and modeling and simulation, defensive information warfare, and intelligent information systems technologies.