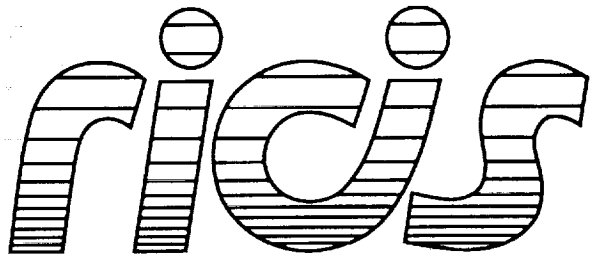
Object Links in the Repository

Jon Beck
David Eichmann
West Virginia University Research Corporation

9/27/91

Cooperative Agreement NCC 9-16
Research Activity No. SE.43

NASA Johnson Space Center
Information Systems Directorate
Information Technology Division



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

INTERIM REPORT

The RICIS Concept

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

The UHCL/RICIS mission is to conduct, coordinate, and disseminate research and professional level education in computing and information systems to serve the needs of the government, industry, community and academia. RICIS combines resources of UHCL and its gateway affiliates to research and develop materials, prototypes and publications on topics of mutual interest to its sponsors and researchers. Within UHCL, the mission is being implemented through interdisciplinary involvement of faculty and students from each of the four schools: Business and Public Administration, Education, Human Sciences and Humanities, and Natural and Applied Sciences. RICIS also collaborates with industry in a companion program. This program is focused on serving the research and advanced development needs of industry.

Moreover, UHCL established relationships with other universities and research organizations, having common research interests, to provide additional sources of expertise to conduct needed research. For example, UHCL has entered into a special partnership with Texas A&M University to help oversee RICIS research and education programs, while other research organizations are involved via the "gateway" concept.

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

RICIS Preface

This research was conducted under auspices of the Research Institute for Computing and Information Systems by Jon Beck and Dr. David Eichmann of West Virginia University. Dr. E. T. Dickerson served as RICIS research coordinator.

Funding was provided by the Information Technology Division, Information Systems Directorate, NASA/JSC through Cooperative Agreement NCC 9-16 between NASA Johnson Space Center and the University of Houston-Clear Lake. The NASA technical monitor for this activity was Ernest M. Fridge, III of the Information Technology Division, Information Systems Directorate, NASA/JSC.

SoRReL

Software Reuse Repository Lab

Object Links in the Repository
Interim Report

Jon Beck & David Eichmann

Software Reuse Repository Lab
Dept. of Statistics and Computer Science
West Virginia University
Morgantown, WV 26506

SoRReL – RBSE – 91 – 1

September 27, 1991

Object Links in the Repository

Interim Report *

Jon Beck & David Eichmann

1. Introduction

This interim report explores some of the architectural ramifications of extending the Eichmann/Atkins lattice–based classification scheme [1] to encompass the assets of the full life–cycle of software development. In particular, we wish to consider a model which provides explicit links between objects in addition to the edges connecting classification vertices in the standard lattice.

The model we consider here uses object–oriented terminology [3, 4]. Thus the lattice is viewed as a data structure which contains class objects which exhibit inheritance.

This report contains a description of the types of objects in the repository, followed by a discussion of how they interrelate. We discuss features of the object–oriented model which support these objects and their links, and consider behaviors which an implementation of the model should exhibit. Finally, we indicate some thoughts on implementing a prototype of this repository architecture.

2. A Bestiary of Objects

The repository is designed to contain the full set of assets created during the software life–cycle. Therefore, there are many types of objects we wish the repository to contain. Listed below are some obvious candidates for inclusion in the repository. This is an open list, indicative but not exhaustive. Extensibility of the system, a strength of faceted classification, is a necessity.

* This work is supported in part by NASA subcontract 089, cooperative agreement NCC–9–16, project no. RICIS SE.43.

Our discussion uses a simplified waterfall life–cycle model solely for the purposes of illustration. Our choice of models for this report was made on the basis of reaching the most general audience, rather than upon the suitability of any particular modeling technique. The arguments presented below apply equally well to any such technique.

2.1 Requirements

A repository containing the assets of a full life–cycle of some software development project will contain one or more requirements documents or requests for proposal which delineate the need which the software met. These documents will be written in human text (possibly with diagrams and figures) but will refer to functionality provided by code.

2.2 Specifications

Based on the requirements, there will be specifications documents, also written in human text. These documents describe the architecture of a software system which will provide the functionality demanded in the requirements. Code is written based upon the architecture which the specifications provide.

2.3 Code

Code is the central category type for the repository. While all the other objects are necessary to a fully functioning repository, code is the repository's focus, and the main attraction for users.

In the prototype stage we concentrate on the Ada language, but extensible support for other languages is essential. Given a grammar or specification for a language, the repository structure must be able to accommodate code in that language.

2.4 Validation and Acceptance Documents, Test Data

After the software has been coded, the development team bears the burden of proving that it meets the requirements and follows the specifications. There can be textual descriptions of how the requirements are satisfied. There can also be files of test input data or script files which demonstrate test cases. There may be files of output data captured to show compliance with the

specifications. There may be caveats listing limitations or implementation dependencies. All of these refer back to the requirements, specifications, and actual code of the software system.

2.5 Versions

All of the above assets may exist in the repository in multiple versions. Version 2.0 of a word processor is very similar to, but distinct from, version 2.1, and it is valid for both versions to exist in the repository. This means that all assets of that word processor package, from requirements to acceptance report, may exist in multiple versions. There could also be a *Differences* document relating one version to the next, which belongs to two versions.

3. Object Granularity

The repository will contain not just code, but code at a number of different levels of granularity. For example, a repository object might be a word processor, available for retrieval as a complete word processing module. But embedded within that package are many other code objects. There might be a queue package for input buffering, which in turn contains a linked list package. The search–and–replace module is an object, but from it can be generated two separate submodules by the technique of program slicing [2, 7], the search submodule and the replace submodule. Each of these is a repository object in its own right, separately retrievable via a query on its own classification.

Similarly, a specifications document for the word processor will exist. But within that document are one or more sections detailing the specification for the search–and–replace module.

A file of test data may be input which exercises the entire package, or it may be input for testing only a very small functional piece of the system. For example, a file containing misspelled words for ensuring that the spell checker functions correctly may have nothing to do with testing the printer output module of a word processing package. However, the file of misspelled words properly resides in the repository as a member of the comprehensive test suite.

Every large object in the repository may contain or be composed of smaller objects also in the repository in their own right. Conversely each small object may be not only a valid repository object but also a constituent of a larger asset.

The issue here is one of complex structure; we use a canonical notion of a document to illustrate the concepts. Consider the general concept of a document with a fixed structuring scheme (sections, subsections, paragraphs, and sentences) as shown in figure 1. Any given document

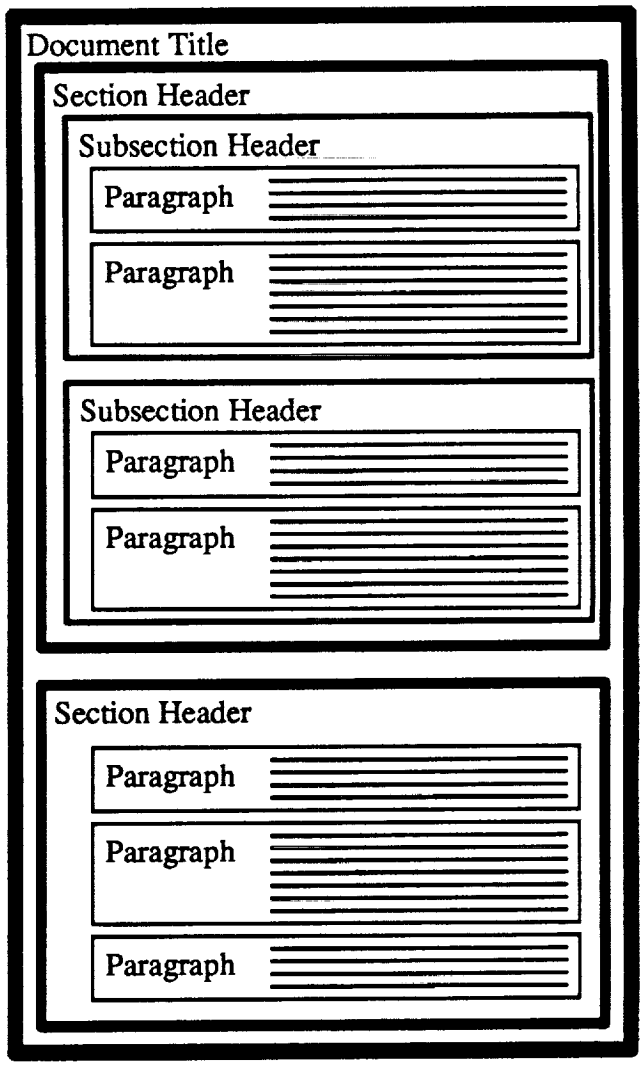


Figure 1. A Sample Document

contains an arbitrary number of sections, which in turn contain an arbitrary number of subsections, and so on.

Every large object in the repository may contain or be composed of smaller objects also in the repository in their own right. Conversely each small object may be not only a valid repository object but also a constituent of a larger asset.

The model includes the definition of the limits of granularity. In the prototype presented here, a *Document*, the coarsest level, contains successively finer objects, down to *paragraphs*, the finest level. The document class definition limits the number of granularity levels. For code, a recursively defined class, there is no fixed number of granularity levels. Every bona fide block in the code, no matter how deeply nested, is a repository object at its own level of granularity. Thus the reference given in section 2.1 for the language's specification to allow parsing code into its block structure.

We do not imagine, however, that each lowest–level object will be replicated in every coarser object of which it is a constituent part. A *paragraph* will not be replicated in every *subsection*, *section*, and *document* which contains it. Rather, the larger–grained objects will contain references to the finer–grained ones, references which are transparent to the user. In object–oriented terminology, the larger–grained objects are composite. More exactly, the references from coarse– to fine–grained objects are shared independent composite references. The reference from a *word processing system* to one of its constituent *string packages* is a shared reference because the string package may be contained in more than one parent object. The reference is also independent because the existence of the *string package* does not depend on the existence of the *word processing system*. We might decide that the word processing system is of no further use in the repository and delete it, but retain the string package on its own merit.

4. Object Links

As outlined above, there are many objects which will reside in the repository. It is obvious that there are many relationships among them. A spell checker code module is related across granularity levels up to the word processing package which contains it and down to the buffer package it contains. It is related across life–cycle phases, back to the specifications section which discusses spell checking functionality and forward to the verification test of the spell checker module. It is related across versions of the software back to its predecessor and forward to its successor.

A person browsing in a conventional library has only one dimension by which to follow links to find related books. From a book of interest, the browser can search left or right along the shelf to try to find related works. But our repository has the ability to provide many dimensions of links to related objects. The basic lattice structure provides two mechanisms for browsing for related objects, relaxation of facet values in queries and use of closeness metrics which produce queries containing conceptually similar or related terms.

In addition to these, the data structure of the objects in the lattice should allow the inclusion of explicit links along all the dimensions given above. These links connect related objects and must be available to the browser as a means to identify objects related along the axes of granularity, life–cycle phase, and version. All repository object links are bidirectional and reflexive. They may be one–to–one, one–to–many, or many–to–many.

The combination of a rich linking structure within a lattice framework produces the potential for an extremely powerful interface mechanism. Traditional relational query systems can only retrieve data blindly, with no notion of their location in the database. Most current object–oriented systems provide only navigational access to data, with limited querying ability. Our model provides full query access to any node in the lattice through the facet–tuple mechanism. But our model also provides full navigational access via the object structure with its cross links. With

this combination of declarative queries and procedural navigation, it is thus possible for the user to browse through the entire repository finding and pinpointing the exact object of interest.

Object–oriented database systems support our link concepts through *object identity*. A reflexive relationship implies that the parties (i.e., objects) to the relationship store the identity (or identities) of the objects to which they relate. This is very similar, but not exactly equivalent, to the concept of pointers in more traditional programming languages.

4.1 Phase Links

Phase links are those which join one object in the lattice to another object which is related by virtue of being the "same" object at a different phase of the life cycle. This type of link joins, for example, a requirement to its embodiment as a specification, and then similarly on to its implementation in code.

There must be a link not only between the word processor's specification document and the word processing code, but also between the section of the specification which treats of the search–and–replace function and the code module which implements that functionality.

Figure 2 illustrates the duality of reference between the various artifacts in the life cycle. A requirements document has as its *specification* some design document (a one–to–one relationship); that same design document in turn was *specified by* the requirements document. A given design document may specify aspects of multiple programs (illustrating a one–to–many relationship).

4.2 Granularity Links

Granularity links are those which join objects across granularity levels. This type of link joins, for example, a *section* in a document is linked to the *paragraphs* it contains, and also to the *chapter* which contains it. Similarly, in source code, a *search* program slice has links to the *search–and–replace* module from which it was derived via slicing.

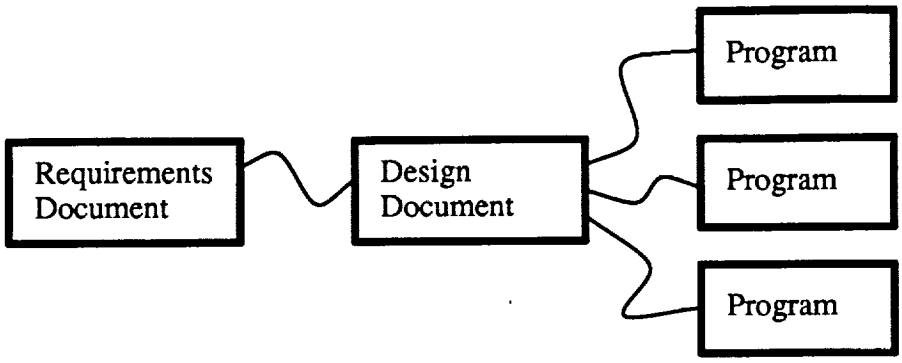


Figure 2. Linkage Between Objects from Differing Phases

The transition from our conceptual model of a document as illustrated in figure 1 to the object model of a document as illustrated in figure 3 exemplifies the representation of complex structure in object–oriented systems.

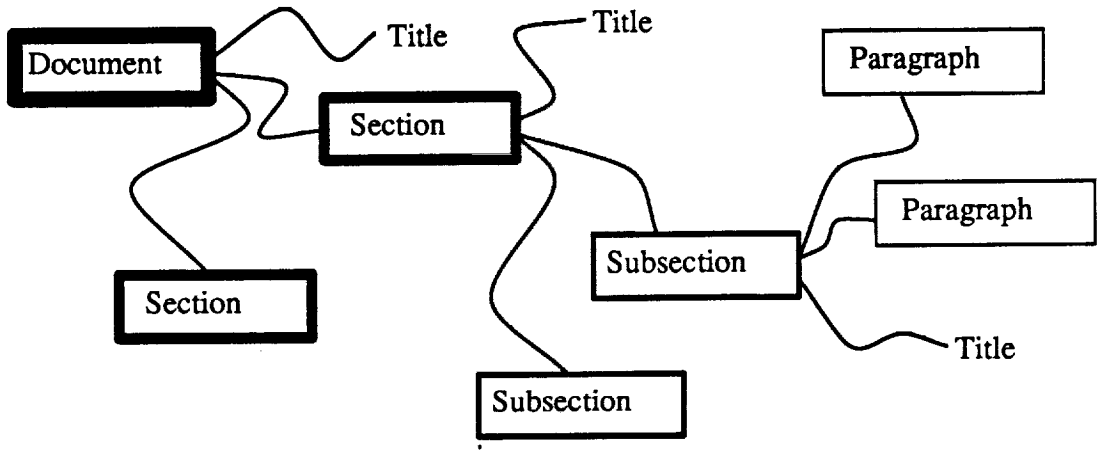


Figure 3. The Granularity References for a Portion of Figure 1

Hence, a document is a title and an ordered collection of sections. A section is a title and an ordered collection of subsections, and so on. Object identity implies that the document does not actually contain all of its nested components, but rather it contains references to them (effectively pointers to the other objects).

4.3 Version Links

If versions are added to the repository, a new dimension is added. In this dimension there are links from an object forward to a later version or backward to a previous version of the same object concept. These links are orthogonal to the phase links between objects in the same project. It is possible, however, that the version relationship is not as simple as lineal descendancy. Rather, the versions of an object may form a directed acyclic graph, as shown by the bold lines in figure 4, designating the derivation of version 2 from version 1, and the derivation of version 3 from both version 1 and version 2. Any number of new versions may be derived from one or more existing versions. In other words, versioning can exhibit all the characteristics of temporal inheritance.

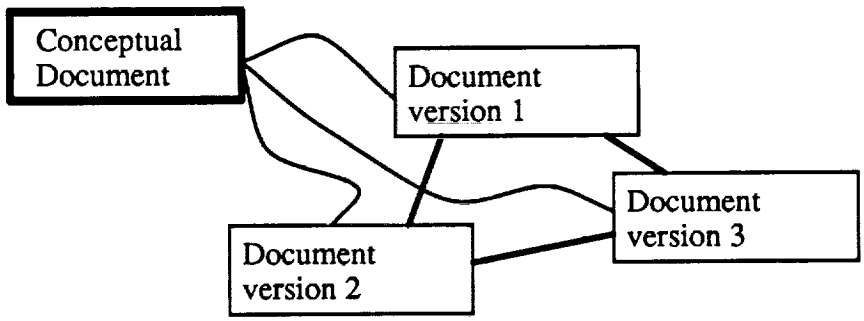


Figure 4. A Sample Multiple–Version Document

The set of versions for some document artifact in the life cycle is just a labeled association, with the version number acting as label for a specific instance of a document object. This leads to the distinction between a conceptual document and a document version. A conceptual document contains the named associations comprising the various versions, each of which are documents in their own right, as shown in figure 4.

Note that any given object can be referenced by any number of other objects, so that it is quite reasonable for a given section to appear unchanged in multiple versions of a document. This is accomplished by storing the identity of the section in each of the documents' respective ordered sequence of sections.

5. The Model

The above sections describe an architecture for a lattice–based faceted repository of life–cycle assets. Many of the features of this architecture are couched in object–oriented terms. We use these terms because the object–oriented paradigm provides semantics closer to the abstract concept we are trying to model than any other yet developed. Use of object–oriented terminology and concepts, therefore, leads us directly into the use of an object–oriented data model for designing the data structures of the lattice.

The conceptual structure of the repository is a lattice, demanding an object–oriented model which explicitly includes multiple inheritance. As depicted schematically in figure 5 and textually in figure 6, the fundamental superclass of the lattice is the LatticeNode class. The two subclasses of LatticeNode are FacetNode and TupleNode, corresponding to the node types in the Facet and Tuple sublattices as explained in [1].

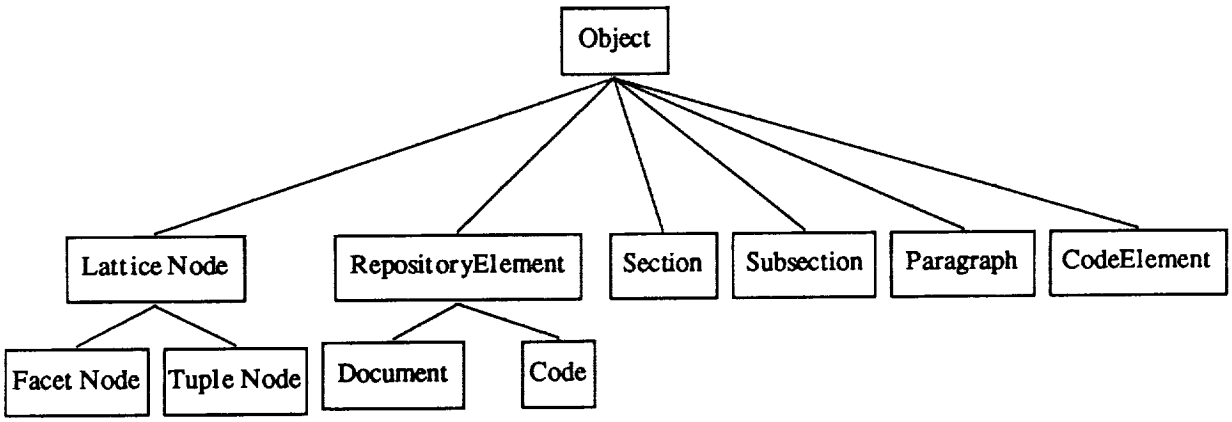


Figure 5. The Class Hierarchy

The Tuple sublattice contains the references to the items actually stored in the repository. An instance of TupleNode contains the attribute *set of RepositoryElement* to accomplish this. In our simplified example, a RepositoryElement is a class with only two subclasses, *Document* and *Code*. In a full repository implementation there would be other subclasses for storing test data and make scripts, for instance.

LatticeNode
    set of LatticeNode — parents
    set of LatticeNode — children

FacetNode : subclass of LatticeNode
    set of FacetValue

TupleNode : subclass of LatticeNode
    set of FacetNode
    set of RepositoryElement

RepositoryElement
    ObjectTitle
    ObjectVersion
    ObjectAuthor
    ObjectDate
    ...other attributes

Document : subclass of RepositoryElement
    ...other attributes
    set of SectionObject — constituent items
    set of FigureObject — constituent items

Section
    SectionHeader
    SectionNumber
    set of Document — parents
    set of Subsection — constituent items

Subsection
    SubsectionHeader
    SubsectionNumber
    set of Section — parents
    set of Paragraph— constituent items

Paragraph
    ParaNumber: Integer
    set of Subsection — parents
    ParaText: String

Code : subclass of RepositoryElement
    CodeLanguage
    ...other attributes
    set of CodeElement — constituent items

CodeElement
    set of Code — parents
    set of Declarations
    set of Statements

Figure 6. The Class Definitions

The RepositoryElement class defines attributes of general interest such as Title, Author, Version, Date. These attributes constitute general metadata about repository object which would be displayed to the user. The subclasses Document and Code have further attributes which are specific to their types. For example, a Document instance might contain a Drawing, whereas a piece of Code would have a ProgrammingLanguage.

As explained in Section 3, a Document in the repository is not atomic but is composed of instances of the classes Section, Subsection, etc. Each of these classes is an object defined with its own appropriate attributes. Similarly a Code instance contains CodeElement instances.

The encapsulation feature of the object–oriented paradigm makes this model easily extensible. For example, if in the future we added to the repository a sound processing program which required a digitized audio score as an initialization file, the requisite class definition of that object could be added to the schema with no disruption of the current existing definitions.

6. Future Work

We have identified the major objects which will reside in the repository and we have proposed an object–oriented data model for our lattice. With this model it is possible to capture the abstract concept of a static lattice repository which exhibits inheritance among its objects and many complex linkages between them. This model also provides for the encapsulation of the functions which allow navigation between and display of the objects in the repository.

We now intend to examine a number of commercial and experimental object–oriented database management systems to determine the feasibility of implementing this model. The result of this examination should be a prototype of ASV4, the full life–cycle reuse repository. We anticipate that this prototyping phase will generate considerable feedback for refining and fine–tuning the object–oriented data model.

Particular areas that warrant further examination include:

- the role of methods (mechanisms that implement behavior) in the presentation of and nagivation throught the repository and its contents;
- the ties between an object–oriented model of the repository and a hypermedia representation of the repository; and
- the assistance an object–oriented model of the repository can provide in quality assessment [5, 6].

References

[1] Eichmann, D. A. and J. Atkins, "Design of a Lattice–Based Faceted Classification System," *Second International Conference on Software Engineering and Knowledge Engineering,* Skokie, IL, June 21–23, 1990, pages 90–97.

[2] Gallagher, K. B. and J. R. Lyle, "Using Program Slicing in Software Maintenance," *IEEE Transactions on Software Engineering,* vol. 17, no. 8, August 1991, pages 751–761.

[3] Kim, W., *Introduction to Object–Oriented Databases,* MIT Press, Cambridge, MA, 1990.

[4] Meyer, B., *Object–Oriented Software Construction,* Prentice–Hall , New York, NY, 1988.

[5] SofTech, Inc., *A Research Review of Quality Assessment for Software,* AdaNet Report ADANET–FD–R&T–086–0, April 30, 1991.

[6] SofTech, Inc., *A Quality Assessment Trade Study,* AdaNet Report ADANET–FD–R&T–086–0, July 12, 1991.

[7] Weiser, M., "Program Slicing," *IEEE Transactions on Software Engineering,* vol. SE–10, no. 4, July 1984, pages 352–357.