A MODEL FOR SUPPORTING MULTIPLE SOFTWARE ENGINEERING METHODS IN A SOFTWARE ENVIRONMENT

DISSERTATION

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of the Ohio State University

BY

William Henry Hochstettler III, B.S., M.S.

* * * * *

The Ohio State University

1986

Dissertation Committee:

J. Ramanathan

D. S. Kerr

W. F. Ogden

Approved by

Jayashree Ramanathan

Adviser
Department of Computer and Information Science

Dedicated to my mother, Edna M. Hochstettler, who was unable to witness the fulfillment of this goal.

ACKNOWLEDGEMENTS

The author gratefully acknowledges the support, dedication and time that Jay Ramanathan willing gave for the completion of this dissertation. Bill Ogden and Doug Kerr, as members of the reading committee, contributed much to the quality and readability of the dissertation and their time is appreciated. The contributions of the TRIAD group, Thorbjorn Andersson, Ron Hartung, Merete Jordal, Walt McKnight, Ronnie Sarkar and Robert Vermilyer did much to inspire and solidify many of the ideas presented in the dissertation. Over the years many colleagues and faculty members have had the belief in my abilities to encourage me to achieve this goal. This list includes: Kevin O'Kane for starting me on computer science research at the graduate level, Paul deMaine, for advising my Master's degree at the Pennsylvania State University, Larry Rose for overseeing my generals examination and Charley Shubra for encouraging me to join the TRIAD Research Group. The assistance provided by the International Business Machines Corporation both in equipment and research funding for the implementation of the TRIAD model is greatly appreciated. Finally, the moral support provided by my parents and the sacrifices made by my wife to enable me to complete this ambition is beyond repayment.

VITA

January 23, 1951	Born Toledo, Ohio
1970–1972	Engineering Practice Program Summers only at Owens-Corning Fiberglas Corp., Toledo, Ohio
1973	B.S., Washington University in St. Louis, Missouri
1974–1975	Programmer-Analyst, St. Louis County Government, Clayton, Missouri
1977	M.S., The Pennsylvania State University, State College, Pennsylvania
1978–1980	Graduate Research Assistant, OCLC, Inc., Columbus, Ohio
1980–1984	Research Scientist, Battelle Columbus Laboratories, Columbus, Ohio
1983	Adjunct Faculty member Franklin University and Capital University, Columbus, Ohio
1984	Applications Programming Manager Health Development Incorporated, Columbus, Ohio
1985–Present	The Ohio State University, Columbus, Ohio

PUBLICATIONS

"Computer Based Records as an Aid to Power Plant Availability Improvement", (Co-author, Don Anson and Larry Stember), Presented at the Joint Power Generation Conference, September 22, 1983.

"A High Level Simulation Model of a Networked Computer System", Proceedings of the 1980 Winter Simulation Conference, (Co-author Lawrence L. Rose), IEEE Long Beach, California, December 1980, pp. 275-289.

FIELDS OF STUDY

The design, implementation and application of practical software environments which can be applied to realistic software engineering problems.

Additional interests include software engineering methods used to support software design, analysis, and construction in addition to the management of the software development process.

In addition to software engineering, operating systems, programming languages, databases and simulation are fields of study.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

This dissertation describes the structure, use and implementation of the TRIAD model, which is a model for the representation and automation of software engineering methods (hereafter referred to simply as methods). The model is designed not only to support the use of single methods, but also to support the cooperative use of multiple methods. In addition, the model is structured so that when a method is described in the model's terminology, computer based support for the model can be readily provided.

1.1 SOFTWARE ENGINEERING METHODS IN THE SOFTWARE LIFE CYCLE

The software life cycle model divides software development into distinct phases--requirements analysis, system design, program design, coding and maintenance [BIGG80]. The tasks accomplished in each phase transform a software system from an idea to implemented code. Beginning with an imprecise idea, each succeeding phase of the life cycle creates a less vague, more precise description of the desired software. At the conclusion of each phase, a document is produced describing the accomplishments of the phase. On the basis of this document, a decision is made on whether to proceed with the development or cancel it. If in the later phases, errors are discovered in work done in the previous phases, the previous phase is re-entered and the errors corrected. The life cycle then becomes iterative. Many iterations may be made through the life cycle before the software implementation is completed and the software distributed for use.

To facilitate software development, many methods have been created which help the software engineer to accomplish the tasks in a particular phase and to manage the overall software development process [DAVI83,FREE77]. In general, methods have two goals. The first goal is to support the building of the software product, while the second is to support the management of the software engineering process.

Methods have three components. The first component is a way of describing the desired software in some particular representation. The second component is a way of describing the meaning of this software representation and the third is a systematic procedure for creating the representation of the software. Methods usually focus on a specific software life cycle phase or on individual tasks within a phase, such as requirements collection or program design. Each method develops a specific representation which is best for representing the software during the phase being applied. This representation is often not the same as (or even close to) the intended final result--the source code.

The methods are usually consistently applied in the initial iteration of the phase; however, if the phase is re-entered, especially for minor corrections, the natural tendency is not to re-apply the method and update the representation in the earlier phase, but just to make the correction in the phase in which it was discovered. Part of this problem is discipline, but the other part is the amount of effort required to maintain the method of a preceding phase during iterations of the life cycle. In addition, the primary focus of the staff is to complete the current phase, not adjust the previous one. This tendency destroys the historical value of the method as a documentation of the software engineering process.

Since the methods are phase specific, different methods are employed in different phases. The transition between phases becomes difficult if the representation for the software is not consistent. One phase may be largely textual, while the next may be graphical and the following hierarchical. In addition, some methods do not even have computer based support.

For those software engineering methods having computer based support for the method application, the support is specific for a particular method such as IDEF for SADT [SOFT81] and PSL/PSA [TEIC77]. Although the use of the method is still beneficial, the expected benefit of storing the method in a computer based support tool is not realized unless the tool has a common representation. Thus, the problem of phase to phase transition is exacerbated by the computer based support rather than lessened.

To support software engineering methods effectively within the software life cycle, a model is needed for representing software engineering methods and providing features to support the method use. This model must be capable of capturing the representation properties of the method as well as the procedural properties. To represent most software engineering methods, the model must be capable of handling many types of data--in particular, large blocks of unstructured text which are characteristic of the early life cycle phases and of software documentation. In the later phases, the model must be capable of representing methods such as Dataflow Diagrams and program structure charts, which are graphical in nature. The procedural properties of a software engineering method describe how tne method is used with or applied to the representational features of the model. Computer based support provides the capability of going beyond merely recording the application of the method to actually assisting the software engineers by reminding them of method constraints and by doing elementary reasoning which may, for example, suggest when design alternatives are possible [WHIL85,YOUR86].

1.2 THE NEED FOR A MODEL TO REPRESENT SOFTWARE ENGINEERING METHODS

Conclusive proof of the value of applying software engineering methods to large projects is inherently difficult to obtain. Experimentation, the obvious approach for proving the value of a method, is too costly to undertake. This is true because experimentation would require developing the project twice--once using a method and then a second time without using one. Another problem with experimentation is an experiment requires the availability of software engineers of demonstrably comparable skill for the parallel developments. So, at present, the only certain thing is that the use of a method is beneficial mainly because the method provides an ordered, reproducible approach to software development. Bergland comments on the motivation for method use as follows, "... software development is so inefficient that almost anything can improve it" [BERG79].

Computer support for software engineering methods can improve the application of most methods. Because of the tremendous processing capabilities of computer systems, the storing, retrieval and processing of the information used by the method can be facilitated. In addition, the computer, which is already needed for code development, can serve as a central information repository for the entire project development. Advanced workstations provide facilities, such as bit mapped displays, multiple windows and keyboard customizations, for all aspects of software development including document production [YOUR86]. A general model with computer based support for representing software engineering methods enhances the use of those methods that do not have existing computer based support and may increase the power of those with computer based support.

In general, methods fall into one of two broad categories; either phase dependent or phase independent [RAMA86]. Phase dependent methods are aimed at supporting the engineer accomplishing the tasks in a particular phase. For example, Jackson Method is aimed at the program design phase of the software life cycle. Phase independent methods are applicable to the process of managing the whole software development process including all of the phases of the software life cycle. Management methods include cost estimation, scheduling and version control. Methods applicable to software in general include traceability (of requirements), metrics and re-usability.

When several software engineering methods are used to develop software, there is a tendency to only actively use the methods while the work is progressing in the phase to which the method is applicable. This tendency causes the value of the method to be lost when changes occur in the software due to errors or new requirements. Thus, the ability to share representations of the software between phases and methods helps the software engineers make changes to the software by easily shifting between phases and methods during the iterations of the software life cycle.

The problem this dissertation addresses is the need for a model for uniformly representing software engineering methods. This model must be capable of capturing the structure of the method, the meaning of the structure, and the rules and procedures governing the use of the method. Since many methods use either hierarchical or graphical structures to represent software and the development process, the model must be able to represent both structural types. The meaning of the structure refers to the implied knowledge inherent in the way the method structures information. For example, SADT makes use of arrows and boxes to represent software. The boxes represent processes or actions while the arrows mean different things depending on their position. Arrows into the left side of a box are input, while arrows from the right side of a box represent outputs. So the model must not only represent arrows, but allow the expression of what the arrows mean in the context of the method. Finally, methods usually have a procedure or a set of rules for the application of the method. To support methods effectively, the model must allow this procedure to be expressed in a form that will aid the user in applying the method according to the rules or procedure. Not only is a language for expressing the rules or procedure necessary, but these procedures or rules must be associated with the proper elements of the method. Using the SADT example again, each arrow has different constraints based on its location on the box. For example, an output arrow, which originates from the right side of a box, can not be attached to the right side of another box. The ability to associate rules and procedures with the method structure is essential to capture the steps necessary to correctly apply a method.

A further requirement for the model is that software engineers find it easy to use both to define and to use existing methods as well as new ones. Finally, the model must allow easy computer implementation in order to support those methods which currently lack such support as well as expand the support for those methods which now have independent or tool based support.

1.3 THE TRIAD MODEL

The TRIAD model is a new model synthesized from research on attribute grammars, databases and knowledge representation systems [TSIC82,DATE77,MINS75,KNUT68]. The TRIAD model provides a framework for capturing the representation of software prescribed by a particular method and for supplying procedures for processing the representation. The processing is provided by specially coded procedures which are associated with the method structure and interfaces to existing tools.

A Jackson Structure diagram can be used to identify and show the hierarchy of elements in the TRIAD model. Figure 1 shows the TRIAD model elements which express the use of a software engineering method. The figure uses a slightly expanded Jackson diagram with a plus (+) used to represent 1 or more instances of an item. The asterisk (*) remains the symbol for zero or more iterations. The 0/1 notation in the bottom right corner of the Refinement Linkages box indicates that the element may have at most one occurrence.

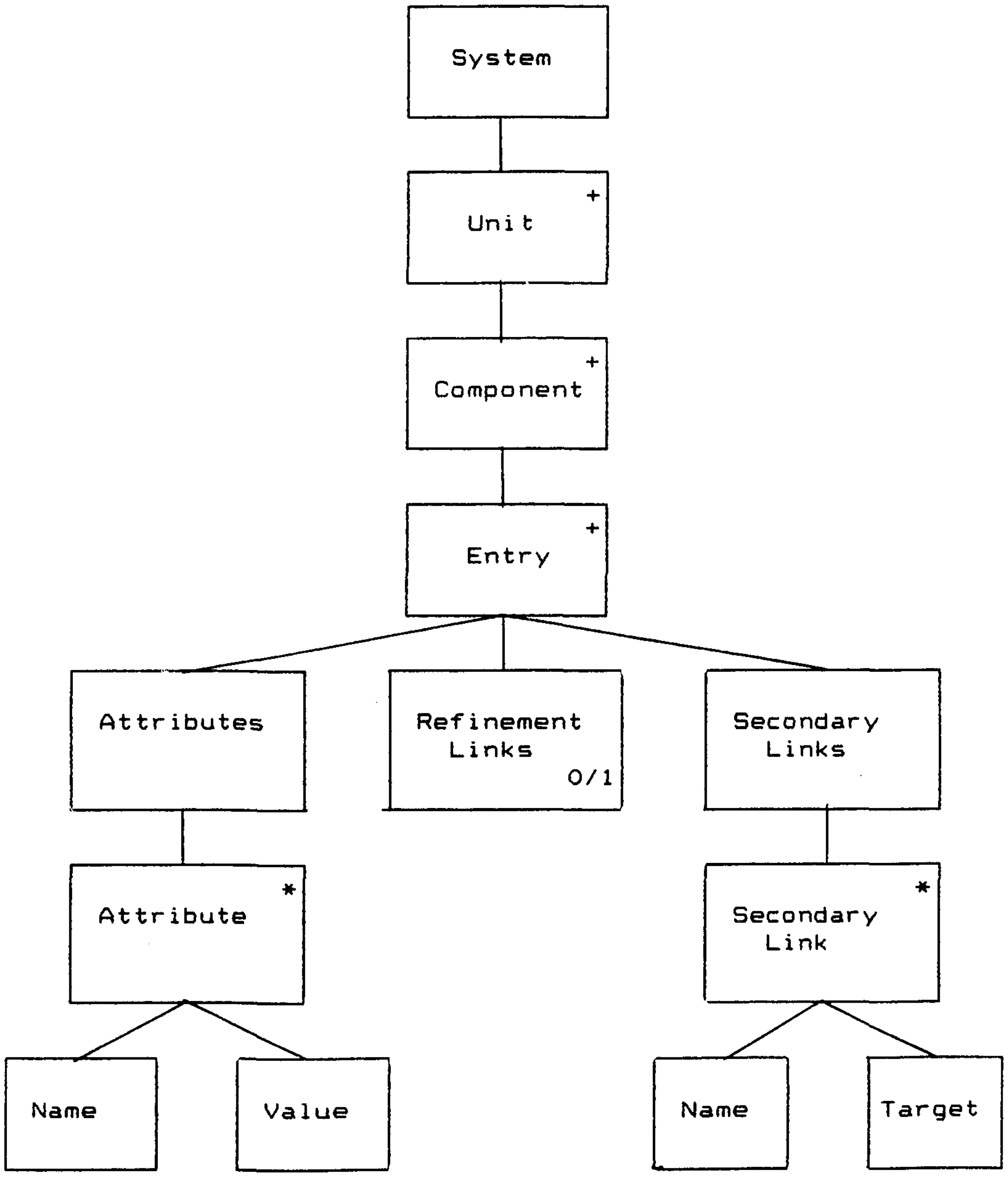


Figure 1. Jackson Method Structure for the Method Use

Typical software engineering methods use symbols to represent aggregates of composite objects. These symbols are often boxes such as those used in Jackson, SADT and Call Structure diagrams, or circles such as in Dataflow diagrams. In the TRIAD model such composite objects are captured by the generic notion of **Units.** Figure 2 shows a simple call structure diagram where each module is represented by a box and the arrows between the boxes represent the source and target of a module invocation. The names of the modules are placed within the box.

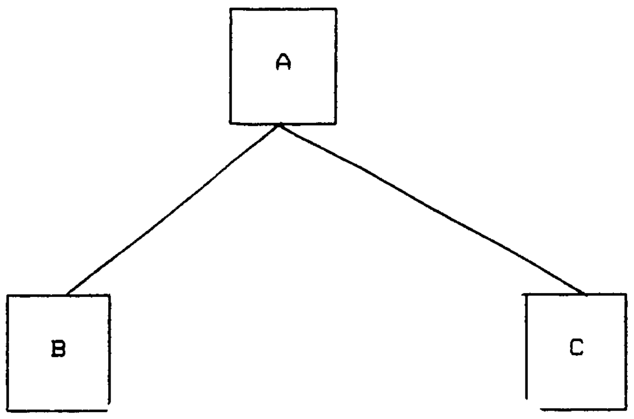


Figure 2. Module Call Structure Example

In the TRIAD model, each module is a **Unit.** The above example represents the structure of a program in terms of module names. The Call Structure method may expand the module description by associating the author's name, date changed, source code and major data structures with each module name. These objects, i.e. the author's name, date, etc., form the composite objects of the method. **Components** represent these objects or Units in the TRIAD model. Table 1 shows the composite objects contained in the Call Structure example.

Table 1. Composite Objects (Units) In

The Call Structure Example

```
Module Name: A
Author: Bob
Date Changed: 09/12/84
Source Code (lines_of_code: 5):
  PROCEDURE A;
  BEGIN
    B;
    C;
  END.

  Module Name: B
  Author: Sarah
  Date Changed: 05/03/85
  Major Data Structure (referenced by C): Z
  Source Code (lines_of_code: 64):
    PROCEDURE B;
    TYPE Z ....
    BEGIN
      ...
    END;

  Module Name: C
  Author: Beth
  Date Changed: 09/12/86
  Source Code (lines_of_code: 128):
    PROCEDURE C;
    BEGIN
      ...
    END;
```

In some methods these Components actually consist of an arbitrarily long sequence of the same object. For example, a module may undergo many changes; therefore, to maintain a history of all of the changes, a list of all of the dates the module was changed is necessary. In the TRIAD Model each occurrence of a Component in the sequence is known as an **Entry**, thus each date a module is changed is an Entry in the "Date Changed" Component.

Entries in turn have various **Attributes** which describe and summarize the Entry. In the Call Structure method example, the Component containing the source code may have an Attribute called lines_of_code which contains the number of source statements contained in the module.

Typical software engineering methods, in addition to using symbols to represent the composite objects in the method, also use symbols to structure these objects. Arrows between modules are used in the Call Structure method to show which modules are called by each module. The TRIAD model calls such arrows **Refinement Linkages.**

Some methods attempt to represent additional relationships between the objects. For example, the Call Structure method may use the data structure definitions for each module to show the common external data elements of the program. In this case, a dashed arrow in the Call Structure method would be used to connect each data structure with all modules that reference it. **Secondary Links** are used in the TRIAD model to show such relationships. The Secondary Links would be from each Entry in a Component containing a data structure to the Entry of the Unit containing the module referencing the data structure.

Figure 1 shows the structure of the elements in the TRIAD model. From this figure it can be easily seen that a Unit is composed of one or more Components. Each Component, in turn, contains one or more Entries. The Entry consists of three elements--Attributes, Refinement Linkages and Secondary Links. The Attributes of which there may be zero or more, contain a name and a value. The Secondary Links may consist of zero or more links also and each link contains a Link Name and Target Entry.

By the use of Figure 2 this informal discussion has shown how a simple method would be expressed using the TRIAD model. Additional features of the TRIAD model permit the generalization of this specific method example to a set of elements capable of representing any program using the Call Structure method. Figure 3 shows the TRIAD Model Method Definition structure which is the generalization of Figure 1.

The **Units** are generalized to **Unit Classes.** For example the Unit Class for the Call Structure method is a single one representing a module. Within each Unit, the **Components** and **Entries** are generalized to **Component Categories** within a Unit Class. The author, source code and date changed are examples of Component Categories from the Call Structure method example. The **Refinement Linkages** are the same except that the source and target are now Component Categories and Unit Classes respectively rather than the specific Components and Units of the method example. The Attributes attached to the Entries are generalized to **Attribute Names.** The Attribute Names include a type definition and the names are associated with a Component Category. Thus, in the Call Structure method example, the lines_of_code Attribute is named lines_of_code, its Type Definition is an integer and it is associated with the source code Component Category. Finally, the Secondary Links are generalized in the same manner as the Refinement Linkages, in that the Secondary Links are named and the source and target Component Categories are named.

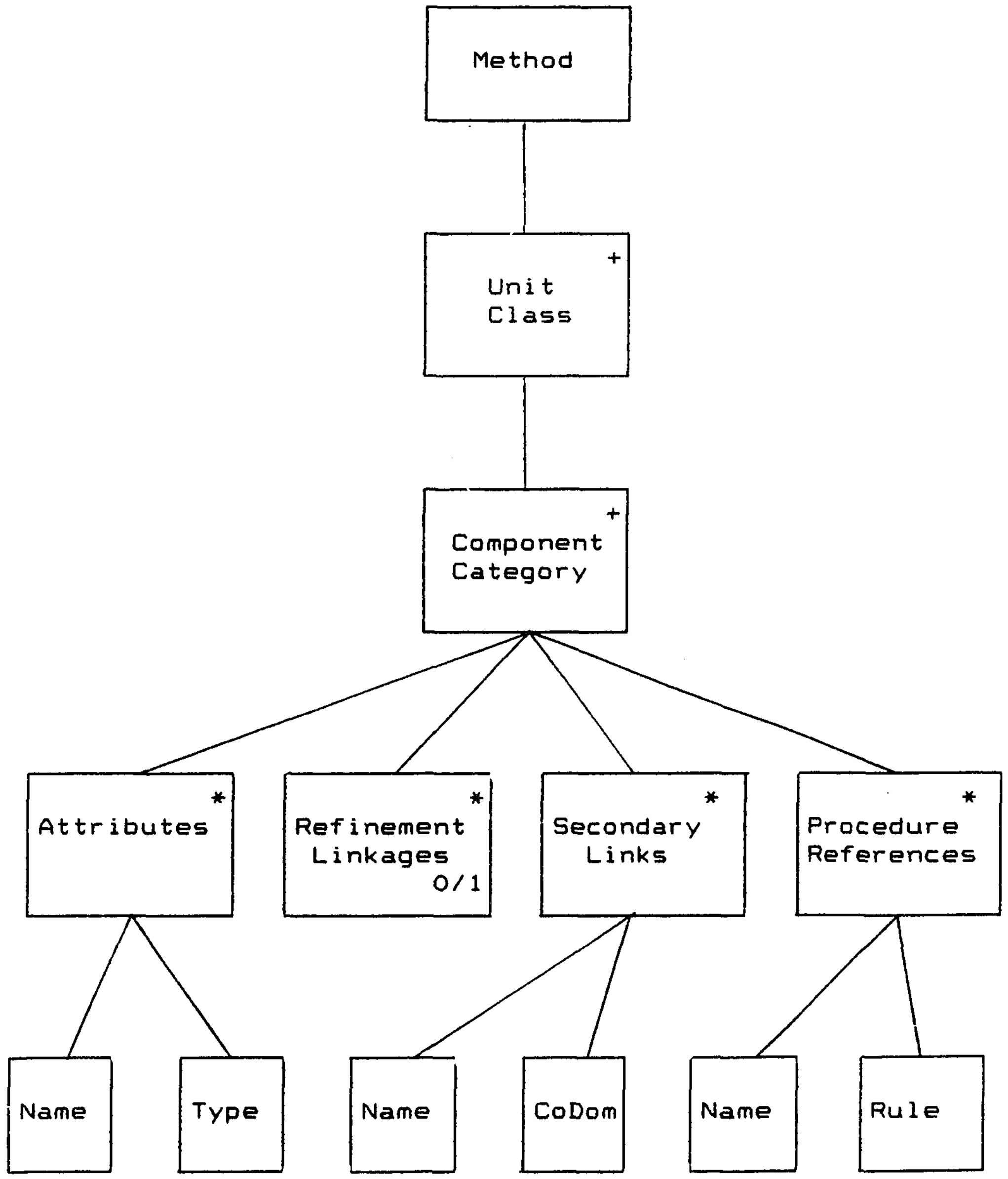


Figure 3. Jackson Method Structure for the Method Definition

Figure 1 shows the generalization of the TRIAD model Method Use Component (shown in Figure 3) to a TRIAD model Method Definition Component. The two figures are very similar. The major difference is the Components and Entries in Figure 3 are generalized as Component Categories in Figure 1 Each Component Category is shown as possessing zero or more Attributes, Secondary Linkages and Procedure References. In addition the Component Category may also contain a Refinement Linkage. Each Attribute has a name and a type. A name and a codomain for each Secondary Linkage is also present. Finally each Procedure Reference has a name and an invocation rule.

Table 1 shown earlier depicted the composite objects contained in the Call Structure method example. Table 2 shows the same objects after performing the generalization described above.

Table 2. Generalization of the Call Structure Method

Composite Objects

```
Module Name:
Author:
Date Changed:
Major Data Structure
    (link: common_ds;
       source: Major Data Structure; target: Module):
```

Source Code (lines_of_code: integer):

The preceding discussion of the TRIAD model has shown how the method's structure can be represented. In addition to the structure, most methods have rules and procedures for putting software into the method's structure. The TRIAD model supports this aspect of methods by allowing procedures to be written and associated with the Component Categories (composite objects of the method). For example, in the call structure example a rule is that each module name must be unique. A small procedure checks each module name as the software engineer creates a Unit for each module against the existing names and ensures that a name is not re-used.

So far only single method support has been discussed, however, software development entails many activities, most of which are supported by different methods. Each method can be expressed separately using the TRIAD model, but the maximum benefit of the iterative nature of the software life cycle is obtained when the different methods are linked together using the TRIAD model.

Returning to the Call Structure method example, this transition to a multiple software engineering method is illustrated when the program design is complete and coding begins, the call structure method can be expanded by adding

Component Categories to support coding such as references to syntax directed editors or Component Categories containing pseudo code.

At the conclusion of the design phase, another different method could be applied for coding support. For this method, the software engineer has several choices. The first choice is to expand the existing call structure method to support the coding process. This expansion can be done by adding Component Categories to the existing Unit Class. In addition, entirely new Unit Classes may be added to support unique aspects of the coding method, which are not already captured in the call structure method.

Another choice is to apply a different method for coding support. Since both methods are defined using the TRIAD model, it is possible to automatically propagate, or in this case, copy the information from the call structure method to the coding method. An alternate approach is to create Secondary Links between Unit Classes in the call structure and coding methods which represent the same module. The ease of transition between methods is possible because a common representation for the methods is used and because the Unit Classes are designed to support the sharing of information between methods.

In addition to the structural representation of the method provided by the TRIAD model, the procedural aspects of applying a method are enhanced by using the model. The definition of the method using the TRIAD model assists the software engineer applying the method by providing a standard representation of the method which when supported by a computer is capable of providing computer based support for the method. Coding of method specific procedures by the method definer to monitor and interpret the method users' actions, provides guidance in applying the method. These procedures can enforce method rules, such as limiting the number of modules called by any other module. In addition, information can be propagated automatically by these procedures. In a management method the completion of the coding of a module may cause quality assurance to be notified, a new test version of the software to be created, and a message sent to the manager that the module is completed which causes a new task for the programmer to be scheduled. All of this is done without any explicit action on the part of the programmer other than indicating that the code is completed.

1.4 ADVANTAGES OF THE TRIAD MODEL OVER EXISTING METHOD SUPPORT

The TRIAD model achieves its comprehensive approach to software engineering method support over the entire software life cycle by focusing on the support of existing methods. The alternative approach is to try to create yet another new method which is applicable to all phases of the life cycle. Support of existing methods is important because the existing methods are widely used and represent a large investment of resources for development of support systems and for training personnel in their use. Thus, uniform computer support is extended to methods in which the current computer professionals are already skilled.

Other attempts at comprehensive method support have used database management systems and attribute grammars to store project data. Database systems do not cope well with unstructured text which is a major component of most methods. Admittedly this is an implementation restriction, but it becomes an issue when computer based method support is provided using off the shelf software. In addition, the underlying data model may not be suitable for the definition of a schema to represent a method. For instance, a relational model can represent a graphical method, but not as precisely or as directly as the network model, which uses a graph to represent the method structure. The issue is how closely does the model reflect the method structure so that the user can easily conceptualize the method once it is expressed using the model.

Attribute grammars were used as the underlying model in earlier versions of TRIAD [MCKN85]. They proved effective, but were difficult to use for those other than computer scientists who are skilled in programming languages. Describing the relationships of the various entities in a typical method requires a great many detailed definitions in the grammar approach. This detail also extends to the Component Categories contained within the Unit Classes. This level of detail is unnecessary because the most important relationship involving Component Categories is that of membership in a Unit Class. For example, in the call structure method, the Unit Class contained several Component Categories (author, date changed, data structure and source code). These Component Category are positioned serially within the Unit Class in the order they were created. In a grammar model, these four Component Categories can be structured in the same manner by the production

A -> B C D E

Categories. An alternate structure for these Component Categories is given by the following productions:

A -> B F

F -> C D E

the grammar model has the expressive power of additional structure for the Component Categories, but the software engineering methods do not require the structure.

The specification of the TRIAD model was driven by two goals. The first goal was to represent existing methods used in all phases of the software life cycle. The second goal was to provide a model consistent with existing methods, such that the software engineer could easily define a method using the model. Each of the major models (attribute grammars, database systems and knowledge representation systems) from which the TRIAD model was synthesized are not capable of satisfying both of these goals completely. Grammars were capable of the representation, but were difficult to manipulate. Knowledge frames were easier to manipulate, but operated at too specialized a level for software engineering methods. Databases compromised on both goals. The representation was not complete, and for some data models it was difficult to manipulate. By selecting and combining the best aspects of all three, a better model was derived.

A key feature of the TRIAD model is the facility for allowing the incorporation of procedures to manipulate and process the representation. This feature will allow method application to become easier by anticipating the software engineer's needs based on the experience of previous users of the method. Without this feature to represent the experience gained in using the method, the relevance of the method is not enhanced or easily customized.

The TRIAD model is consistent with software engineering methods because it supports graphical connections and text storage. The majority of the methods rely on graphical models; especially to represent the software code. The inclusion of a graphics interface allows a symbolic manipulation of the model (entries and categories), thus providing the software engineer with an even more consistent representation of the method.

The implementation of the TRIAD model demonstrated that the model was easily automated. It also made it easier to validate the model by supporting rapid and accurate application of the model to a number of example methods. In addition, the implementation process and the use of the implementation suggested improvements in the model. One of the results was the creation of special features to support classes of methods. Most of these special features are unique Attributes and Procedures which control the presentation and use of the Component Categories within the Unit Classes.

The best demonstration of the value of the TRIAD model was its use to describe a multiple method environment. This exercise went beyond just specifying the several methods; the multiple method was actually used to apply the software engineering methods to the creation of a new version of the TRIAD model implementation. As with the other uses of the implementation, significant insight was gained into the use of the model and into the improvement of the TRIAD model implementation.

The TRIAD model, because of its capability for representing methods, can be used in an evolutionary way. If a software project has already begun or has ever progressed as far as the maintenance phase, it is still possible to apply a method without expending excessive effort to reformat the previously acquired information. This capability was demonstrated by applying the TRIAD multiple method environment to the TRIAD model implementation after the development of the TRIAD environment generator had already begun. If references to software source code can be easily isolated from existing sources, say compiler control statements or even the source language statements, then instances of the method Units which represent modules can be created automatically. By automatically creating the Units, the method is applied even though it may be in a superficial way. In the future, as the code changes, the appropriate Units can be filled in. Over a period of time many of the modules would be completely expressed in the method using this technique of applying the method fully only to those elements of the software which are being reworked. Although the effect of this approach may be only local to the modules being actively worked on, it is still a way to incrementally apply a method without undue startup overhead.

1.5 CONTRIBUTIONS

This research contributions of this dissertation are:

- o Specification of a single model for representing multiple software engineering methods in a software life cycle development process,
- o Implementation of the TRIAD model for proof of concept demonstration,
- o Evaluation of the model and its implementation for multiple software engineering methods support and

o Refinement of the TRIAD model through the creation of a software engineering method consisting of multiple methods to support the development of a large software project.

1.6 ORGANIZATION OF THE DISSERTATION

The dissertation proceeds from an examination of existing software engineering methods, of their current computer support base and of their shared features which must be integrated if they are to be used in a cooperative way within the software life cycle to a proposed model for representing software engineering methods. Current research is surveyed to isolate important features for the construction of a suitable model for software engineering methods. The TRIAD model is implemented and demonstrated using a multiple software engineering method derived from the process of implementing the TRIAD model. An examination of the results of the research concludes the dissertation.

Chapter II explores the general nature of software engineering methods by describing several popular methods. The state of computer based support for these existing methods is also discussed. From the survey of these methods, the requirements necessary for computer based support within the context of the software life cycle is presented.

The TRIAD model is defined in Chapter III. The features of the model as applied to software engineering methods support are described in Chapter IV. Multiple method support features of the TRIAD model are also described in Chapter IV. Chapter V examines alternative models and establishes why they are not as effective as the TRIAD model for representing software engineering methods.

In Chapter VI the implementation of the TRIAD model for the TRIAD environment generator is described. Use of the TRIAD model features for software engineering methods is illustrated by citing examples from the implementation.

A sample multiple method software engineering environment generated by TRIAD is described in Chapter VII. Chapter VIII concludes the dissertation by evaluating the TRIAD model and its implementation.

CHAPTER II

THE NEED FOR SOFTWARE ENGINEERING METHODS

As the cost and complexity of software development has increased over the years, software engineers have been searching for ways to manage the construction of software such that a quality product can be built within schedule and budget constraints and which satisfies the user. Software written twenty years ago consisted of small programs which ran on small expensive computers. The cost of the hardware far exceeded the software development cost. However, now the reverse is true. The cost of hardware has plunged while its capacity has greatly increased. Further, more complex problems are now being attacked because the computers are more powerful. Software engineer's salaries have increased not only because of the inflation of the past decade, but also because of the still chronic shortage of good software engineers. High labor costs and bigger more complex software have been the major contributors to the now higher development costs for software [BOEH84, YOUR86].

To effectively manage these growing costs and produce a quality product, software engineers turned to methods to organize, assist and simplify the software development process. Methods were intended to do the following:

- Provide an ordered way of accomplishing a software engineering task, thereby, moving software development from an art to a science,
- Organize the information produced from the software engineering task for easier processing and retrieval,
- Describe the software engineering problem and solution completely, succinctly and unambiguously,
- Suggest solutions to the software engineering problem. This aspect of a method takes advantage of previous experience when a new problem is recognized as similar to an older, already solved one,
- Produce good solutions,
- Produce solutions faster than not using a method and
- Provide a basis for managing the software engineering problem solving process. By using an ordered approach, progress can can be quantitatively measured and the process properly managed to insure reliable software is produced on time and within budget.

2.1 SOFTWARE ENGINEERING METHODS

Over the past decade numerous software engineering methods have been proposed to assist the software engineer in building quality software. Several of the more popular methods have been analyzed to obtain the requirements to provide computer based support for these methods. Five methods will be briefly described (SADT, Data Flow Diagrams, Call Structure Charts, Jackson Method and Flowcharts). A single example will be used to illustrate the salient features of all five methods.

A simple data processing application is used to illustrate the software engineering methods. The example software is a name and address file with the following requirements:

1. Edit new name and address transactions,
2. Update the name and address file and
3. Produce reports and mailing labels.

Figure 4 shows how the high level processing of this example is expressed using the Structured Analysis and Design Technique (SADT) [ROSS77b].

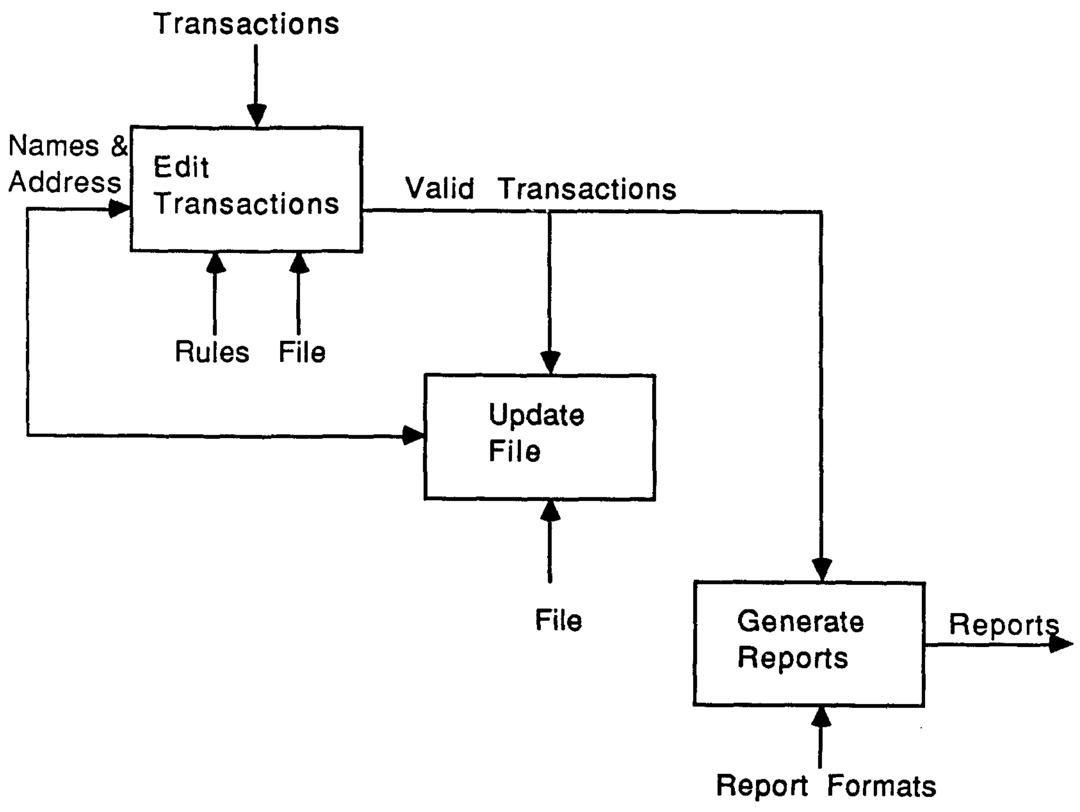


Figure 4. SADT diagram of the Name and Address File System

Each box in SADT terminology represents a bounded context. In this example, the bounded context is a processing action. Arrows into the box from the side are input, which in this example, the input is names and addresses. Arrows out of the boxes are output. Arrows into the box from the top are controls, which are transaction types in this example. Mechanisms are represented by arrows into the box from the bottom. In this example, rules for editing and the file of names and addresses are mechanisms used to edit transactions. Each box and arrow is named with a descriptive label. The purpose of SADT is to communicate ideas which in this case is a software design. No more than 6 boxes are permitted on a single SADT drawing. If more than 6 boxes are needed, than the drawing must be hierarchically organized. Each box in a drawing may be expanded by creating a new drawing containing more detail. Returning to the example, the second box, Update File, could be expanded and all of the processing actions for each transaction described on another SADT diagram.

Figure 5 is the name and address example defined using the dataflow technique [DEMA79]. Dataflow Diagrams represent software by showing the flow of data through processing actions. Bubbles (circles) are used to represent a processing action and arcs between bubbles represent the flow of data. Rectangles are used to represent sources and destinations of data. A data store is represented by the open ended rectangles. Labels are placed in the bubbles and rectangles and on the arcs to describe them.

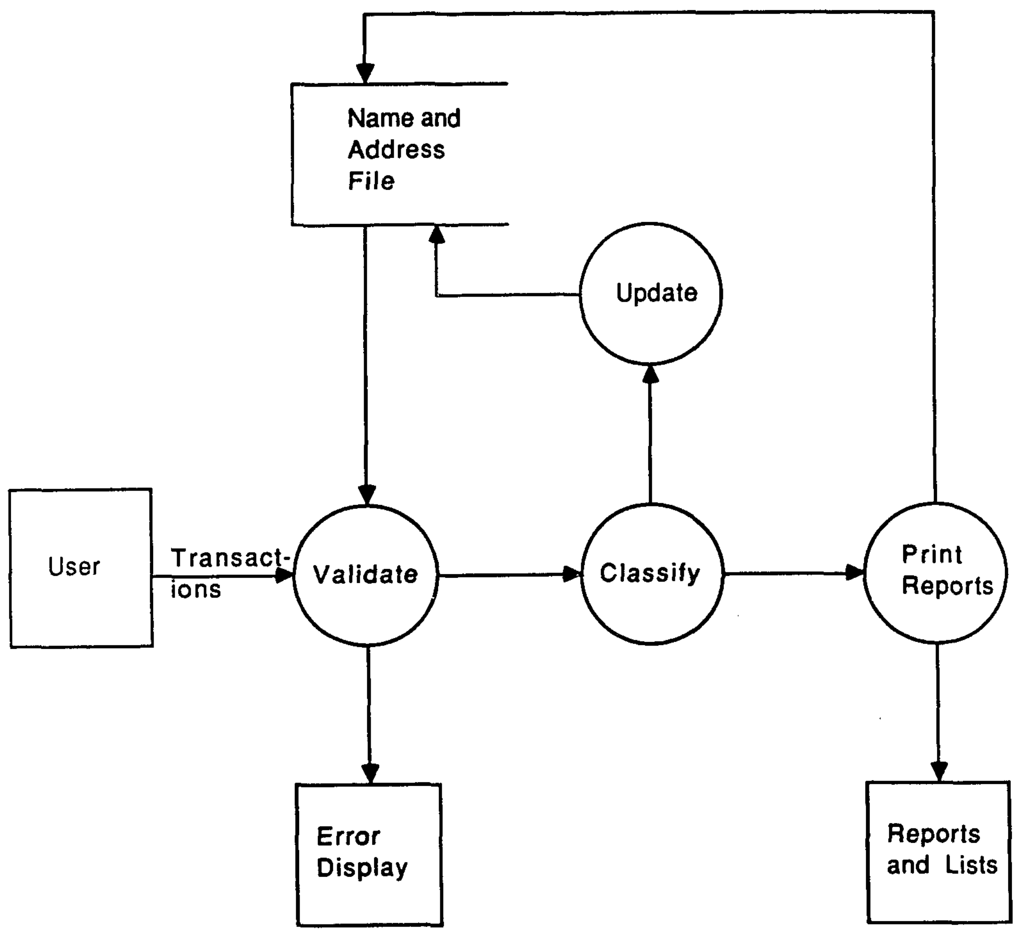


Figure 5. Dataflow Diagram of the Name and Address File System

Referring to Figure 5, transactions flow into the validate bubble and are separated into valid and invalid ones. The invalid ones are displayed for correction, while the valid ones are separated into file update requests. Depending on the transaction type, file updates are made, otherwise the requested reports or labels are printed.

The Call Structure method shown in Figure 6 shows the organization of the example into program modules. The main program calls three submodules, Edit, Update and Report. Each module is represented by a box. Arrows between the boxes represent the calls relation between the modules.

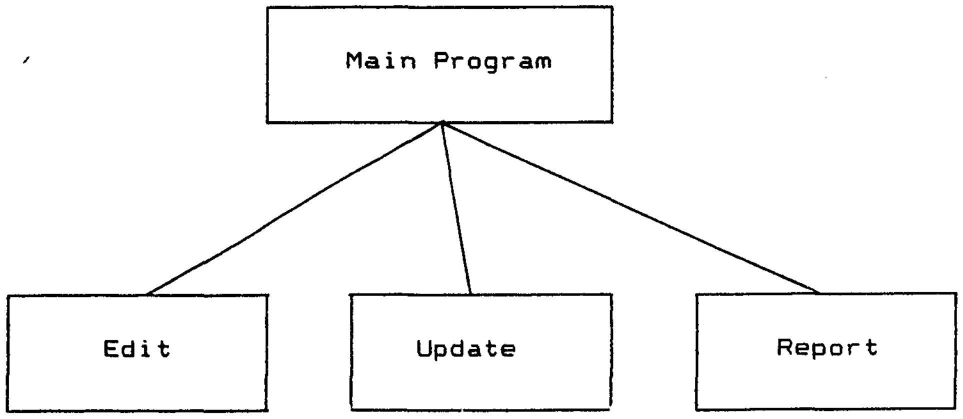


Figure 6. Name and Address System Call Structure

The Jackson Method represents the name and address system as shown in Figure 7 [JACK78]. Data structures are designed first in the Jackson Method and then used to define the processing. The example system has a transaction and data store as the primary data structures. Rectangles are used to represent processing in the Jackson Method and lines between the rectangles represent control paths. Within the rectangles are labels to describe the processing. Three different types of processing are represented in the Jackson Method by slight modifications of the basic rectangle. If a star (*) is placed in the upper right corner of the rectangle then iteration is represented. The processing indicated within the box is repeated until a stated criteria is met. Iteration includes the programming constructs of DO and REPEAT. Selection (choice) is represented in Jackson Method by a zero (O) in the upper right corner of the rectangle. Each selection box represents one of several choices. The IF statement in many programming languages is used to implement the selection construct. Finally a box with no special symbol in the upper right corner is a processing action that is performed in sequence. The sequence of operations is determined by reading the diagram top down and from left to right.

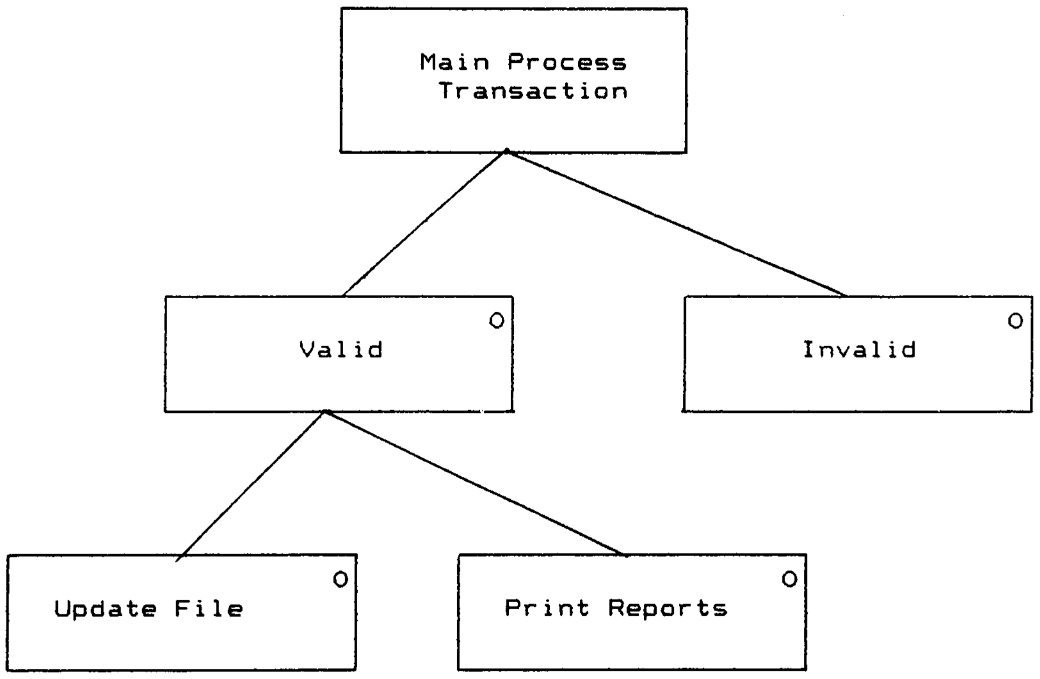


Figure 7. Jackson Method Representation of the

Name and Address System

In Figure 7, selection is used to separate the valid from the invalid transactions at the first level and again at the second to separate the file update transactions from the report transactions.

The final method presented is the Flowchart. Figure 8 shows the name and address file system main processing loop. Flowcharts use distinct geometric symbols to represent processing options and storage entities. The symbols are connected by arrows which represent the flow of control through the symbols. The box represent general processing. Diamonds are decisions and cylinders represent storage entities. Contained within the symbols are descriptions and names for the actions represented by the symbol. For example, in Figure 8, the decision diamond contains the test conditions.

In the initial description of the name and address example, the most common software engineering method was used, namely natural language narrative. The requirements of the system were specified as a simple list.

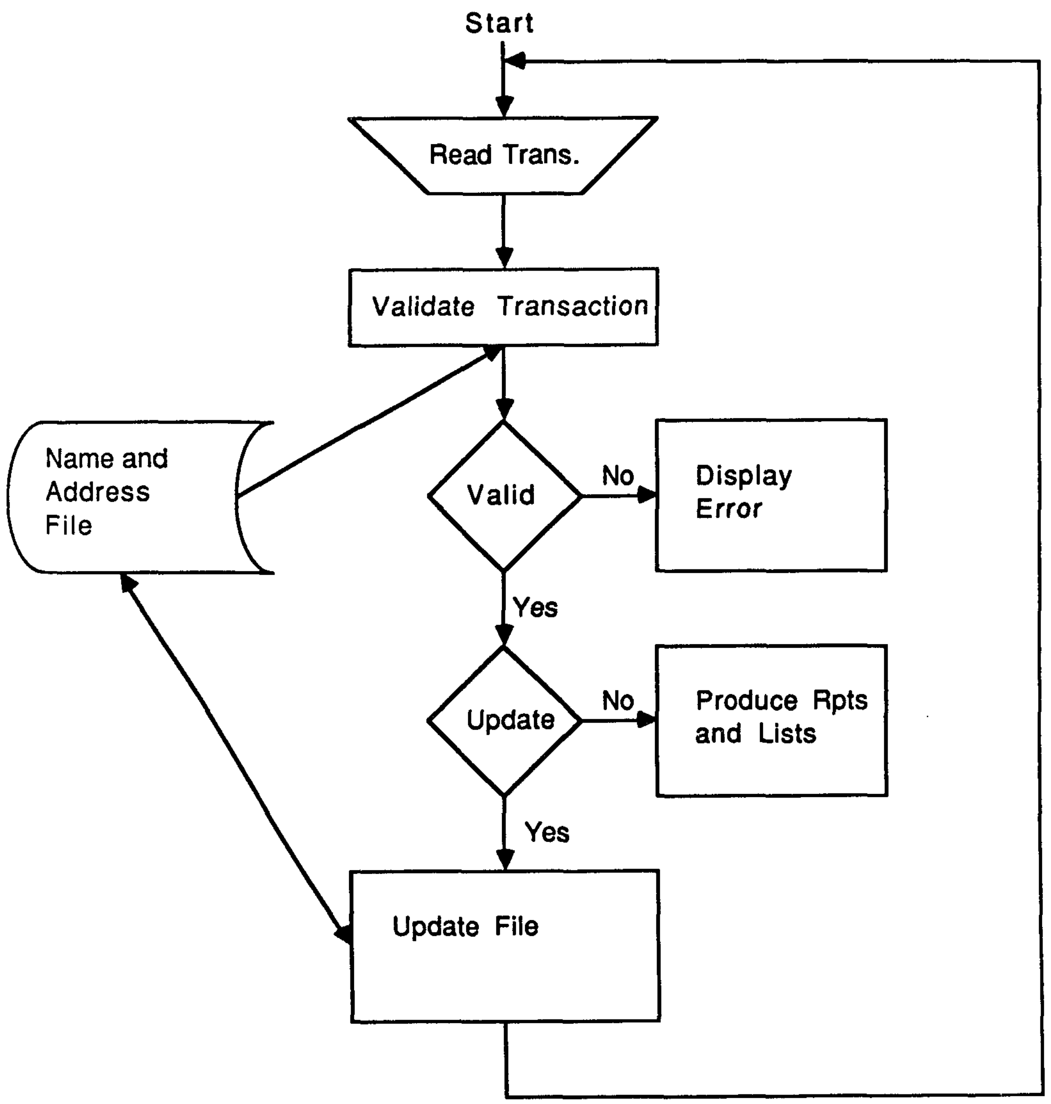


Figure 8. Flowchart of the Name and Address File System

From this brief overview of several popular methods, the following common properties concerning the representation of methods emerge:

o Methods may be entirely textual,

o Methods may combine text and symbols and

o Methods may use graphs to represent the structure of software.

Software Engineering methods are used more as a representation of a solution than an actual problem solving procedure. For example, Dataflow diagrams [DEMA79] and Call Structure Charts [DAVI83] represent the flow of data through a system or the Call Structure of a program, respectively. As a representation of the program, they are effective in providing an exact description of the problem. Ross makes the point about SADT [ROSS85], that the SADT diagrams serve as a documentation of the software for review and agreement by the project participants.

In addition to representing the form of the software, methods also serve to describe it. Data flow diagrams name the source, destination and the path for data elements. They also allow descriptive information about the processing to be recorded within the bubbles.

Other information about the model is also recorded in some methods, such as creation date, revision name, designer, etc. This data is important to manage the use of the method and describe the process of applying the method.

Finally, some methods go beyond representing the program and actually assist the software engineer by suggesting solutions or designs. Jackson Method [JACK78], when properly applied, produces a design, rather than just recording the representation of a design.

2.2 METHODS IN THE SOFTWARE LIFE CYCLE

The development of software is generally viewed as an iterative process consisting of several phases. Although many software life cycles have been proposed consisting of differing numbers of phases, the key idea is to partition the software development process into distinct phases [BIGG80]. These phases have a distinct beginning and ending and produce a document or product whose quality can be evaluated and used as a basis to make a decision on continuing the software development. A general definition of the software life cycle consists of the following five phases:

- o Requirements Analysis - Software development is initiated by specifying the requirements the proposed software is to satisfy.
- o System Design - An overall design of the software is created to meet the requirements defined previously.
- o Program Design - The system design is further decomposed

into programs where the processing detail is specified.

- o Coding and Testing - The program design is translated into a computer language and the resulting code is tested.
- o Maintenance - Errors in the design and coding are corrected.

The software life cycle is a convenient vehicle for classifying methods. The first methods created were those to support the coding and testing phase. This was probably because the coding process was the best understood phase and also the easiest to support by computer tools since the program source is stored in machine readable form. Example methods for program coding include Flowcharting [DAVI83], Structured Programming [DAHL72], Pseudocode [DAVI83] and indentation techniques. The program design phase is supported by methods including Jackson Design Method, Logical Construction of Programs [DAVI83], and Modular Design (both top-down and bottom-up). Methods such as SADT [ROSS77a,ROSS77b], Logical Construction of Systems, PSL/PSA [TEIC77], Data Flow Diagrams [DEMA79], Gane and Sarrenson Charts [DAVI83] and HIPO [DAVI83] were created to support the system design phase. The maintenance phase may use all of the above software engineering methods since it is during this phase that errors in design and coding are corrected. The requirements analysis phase is supported by SADT and SREM [ALFO85]. Table 3 summarizes the many methods by software life cycle phase.

Table 3. Software Engineering Methods

by Software Life Cycle Phase

- o Requirements Analysis
  - Requirements Statement
  - Software Requirements Engineering Method (SREM)
  - Structured Analysis and Design Technique (SADT)
- o System Design
  - Problem Statement Language/Problem Statement Analyzer (PSL/PSA)
  - Hierarchy plus Input/Processing/Output (HIPO)
  - Structured Analysis and Design Technique (SADT)
  - Data Flow Diagrams (DFD)
  - Logical Design of Systems
- o Program Design
  - PDL
  - Jackson Method
  - Structured Design
  - Logical Design of Programs
- o Coding and Testing
  - Structured Programming
  - Pseudo Code
- o Maintenance

In addition to meeting the requirements for methods stated in the previous section, these methods have several common features:

- o Limited Form - Most of the methods use either graphical representation (Flowcharts, DFD and SADT) or a precise language (SREM and PSL/PSA) to organize the information in the method.
- o Reflect the structure of the software - This is particularly true for the system, program design and coding phases.
- o Most of the methods support the development of the software - In addition the methods provide information about the process of software development. Other uniquely management oriented methods such as PERT, CPM and Gantt charts support the process of software development management directly.
- o Most of the methods use a combination of textual and graphical data.
- o The methods can be supported either partially or totally by computer based tools.

2.3 COMPUTER BASED SUPPORT FOR SOFTWARE ENGINEERING METHODS

Many methods have no computer based support, which makes the application of the method different from the majority of the work done in the software engineering process, especially code development and testing. Since the majority of the code development is done using a computer, methods that can be used on a computer simplify the software engineering process by providing a common access mechanism. Further, computer based methods can take advantage of already recorded information.

Several current trends indicate that computer support for all methods is possible:

- o Use of work stations (terminals or personal computers) to do coding. The availability of ready computer access encourages the use of computer based methods,
- o Availability and use of word processing software and high quality printers to do documentation and reports. This characteristic obviously encourages the storage of all project related data on the computer systems, making the use of methods for the text based phases more accessible,
- o The availability of high resolution graphics on the work stations encourages the support of methods which employ

graphical representations for organizing the information contained in the method and

o Sufficient on-line storage to store large amounts of textual data. For a collection of methods applicable to all phases of the software life cycle, there must be adequate storage to store all of the information on-line as well as accommodate indexes to properly organize the information.

These factors are necessary to construct a practical computer based support package for software engineering methods. Without a workstation for ready access to a computer with the above characteristics, computer support for methods is not helpful to the software engineer. The computer acts as a central focus for the entire project and makes it easy and natural to use computer supported software engineering methods.

Existing computer based support for software engineering methods is of two distinct classes, isolated tools and method specific software. Tools by definition are general purpose, single use utilities such as pretty printers, sorts and searching programs. The best example of the tool approach is the Programmer's Workbench on UNIX [DOLO78]. Sharing a common file system, this tool collection works well for specific operations. Complex operations require either parameters on the tool invocation, the coupling of more than one tool together using the pipe or user interaction controlled by the tool. If the tools are not designed to use a similar interface then tool use becomes difficult.

Method specific software packages are just that--specific to the imbedded software engineering method and not capable of being used to represent and apply other software engineering methods. For example, SADT is supported by IDEF, which is an editor and storage facility for SADT diagrams. Tools such as IDEF store the method representation of the software in a unique internal format. To access the method specific information available involves writing new tools or coding an interface to translate the information from the internal format to another standard one. Further, some of these method specific tools do not have an open architecture to allow the interfacing of external tools. It would not be feasible to apply tools such as IDEF which is specifically designed for SADT, on other methods such as Jackson or dataflow.

Methods, such as PSL/PSA, use database management systems to store and manipulate the information in the method. For many methods, databases are unsuitable because they are designed for fixed format, transaction based processing. [KENT79] Even though a database may use a graphical data model such as the hierarchical or network model, it is often incapable of displaying the data graphically. Since many software engineering methods are graphical representations, such as SADT, Dataflow Diagrams and Call Structure charts, a graphical view is essential.

Syntax directed editors support a class of editing methods. Editor generators, such as ALOE, allow the user to specify the language syntax, for which the ALOE creates a structure editor. Any text entered using a structure editor is stored in the form of a parse tree. Action routines provide a means for implementing constraints on the language entry and do syntax checking. The problem with ALOE and other syntax directed editors, is that their support is primarily of one phase--coding. If a software engineering method is not in the form of a language then the method cannot be directly specified. Another drawback to structure editors is that the person doing the editing must constantly think of the text in terms of the parse tree imposed by the language for which the editor is designed. For complicated languages, complicated structures will result, making the editing process more difficult. Further, the structure of the text may not be as important as the content of the text.

Support for software engineering methods requires computer based support beyond isolated tools or support packages. The computer based method has to keep track of the software engineer's actions and be able to relate different pieces of information together to assist the software engineer.

The following features are necessary for computer based support of software engineering methods:

- o Storage of information,
    - Fixed format,
    - Textual,
    - Graphical,
- o Represent the structure of the method,
- o Capture the meaning of the method structure,
- o Provide extended commands to do custom processing,
- o Control access to stored information,
- o Allow multiple user access and protection of information,
- o Maintain versions of method applications and
- o Interface to existing tools.

The initial benefit of applying a method is the organization of the information into a structure that can be analyzed and used as a basis for communication between project members. To this end, computer based method support must be flexible enough to support different types of methods. Support must be provided for methods that are largely collections of text for documentation, requirements specification or module processing descriptions. Fixed format data support is necessary for methods that collect management data such as time and cost expenditures. Finally, representation of graphical methods is required to support the software engineering methods that represent designs and project progress as graphs. For instance, system structure, Dataflow Diagrams, Jackson, SADT and PERT charts all use graphical representation.

The representation of software engineering methods should be general enough to not only be re-usable for different and new methods, but also to allow customization and refinement of the method as experience is gained while applying the method. For example, IDEF is a customized SADT method applied to manufacturing problems. The representation used by a computer based software engineering method should closely resemble the model the method uses to represent the software or the process of software construction. This is important from a human engineering standpoint. If the computer based support uses a graphical representation, then the graphical methods can be easily represented. Further, the software engineer applying the method will not have to translate between the method representation and the support representation for the method.

Going beyond just representing the structure of the method, the model should provide the means for capturing the meaning of the structure. For example, Jackson Method specifies three different types of processing boxes (sequence, iteration and selection) which are distinguished by symbols placed in the upper right hand corner of the box. The method designer should be capable of differentiating between the structures and also the meaning of the structures. Selection involves choosing only one of a series of boxes which are subordinate to the predecessor box while sequence performs the processing actions of each subordinate box serially.

Extended commands facilitate the customization of computer based support for methods. By allowing the software engineer to implement extended commands, it is possible to anticipate the processing needs of the person applying the method. In addition, extended commands implement processing which is peculiar to a particular method the processing can be invoked by a single name.

In addition to the extended commands, the computer based support must provide a query language to retrieve, display and reformat the information organized by the software engineering method and stored by the support package. Queries can be either built on demand from primitives or predefined and stored as extended commands. The software engineer applying the method, invokes the extended commands by specifying the command name.

Besides the organization of information describing software, a method also contains the steps for successfully using the method. Therefore, the support package should include a means for writing instructions to guide the software engineer applying the method. Guidance can take the form of restricting access to information in parts of a method until preceding steps are properly concluded. For example, the coding may not begin on a module until its design is completed. The need for this feature varies from method to method and the restrictions on access must be specified when the software engineering method is defined. Further guidance may require the method applier to completely fill out all descriptions of the symbols before defining another processing action.

The majority of software engineering methods are aimed at large software projects, which have several software engineers working together, thus, the support package must allow multiple user access to the method and its information. At the same time, to maintain the integrity and privacy of the data, sufficient controls must be enforced. This problem is identical to the access problem in database management systems and is therefore solved by making use of the solution for database management systems.

The construction and management of software is an iterative process. Not only is it iterative, but often it is necessary to backtrack and return to an earlier design, plan or code implementation. To support this feature a support package must allow different versions of a method application to be maintained and easily retrieved for examination.

To avoid extensive recoding of tools for a method application, a flexible interface to existing external tools is necessary. This implies that the environment must have an open architecture for its storage and retrieval mechanism. It must also supply processing primitives for accessing the storage facility and a convenient way for invoking the external tools that are interfaced into the environment.

The advantage of using a model that facilitates computer based support for methods is the ability to represent most methods--existing and new. In addition, the common interface provided by the method support package minimizes the amount of effort involved in applying a new method. The alternative of providing method specific computer support for each method, is to create a different interface for each method used. This approach would complicate rather than facilitate the use of multiple methods on a project.

2.4 ENVIRONMENTS TO SUPPORT SOFTWARE ENGINEERING METHODS

An environment is more than just a synonym for the computer and its operating system. An environment encompasses everything affecting the users' work. This includes the lighting, temperature, furniture, hardware and software characteristics. In this dissertation, environment will mean the software, which includes the operating system, the text editor, file system, utilities, tools, database management system and any other software the software engineer uses to accomplish his work.

The necessary elements for an environment for software engineering method support are:

o An editor for manipulating text,

o A storage and retrieval mechanism for access to the information collected during the application of a method,

o A model for representing the method which is flexible enough to represent the structure embodied in many different methods (hierarchy, network and directed graphs),

o A consistent interface to all software engineering methods to minimize the effort required to learn the use of a new method,

- A tool interface,
- Support for the enforcement of software engineering method application and
- Support for project control and management activities.

Of these elements, most are dependent on the host computer system. Even the text editor should be or resemble the one available on the system to minimize the amount of training necessary for new users of the environment. The storage and retrieval mechanism is not directly used by the user except for the query language. Therefore, the most important requirement for an environment to support methods from the view point of this dissertation is the model used to represent the method. It must be general enough to represent most if not all methods and yet be capable of being tailored to represent specific methods easily.

Since most methods are primarily used to represent software, an appropriate model for methods must be capable of representing software structures. The model must capture the organization of the concepts in a way that is consistent with the method. For example, if the method uses a network to represent the Call Structure of a program, then the model must support the representation of networks.

Environments to support methods differ from tools and method specific support packages for methods primarily in scope. The environment provides support for all aspects of the method while a tool may provide support only for a single activity. Pretty printers are good examples of tools. They reformat an existing collection of text. However, they do not support the editing or entry of the text, nor do they provide assistance in the interpretation of the text. Method specific packages only support one method.

It is important to differentiate the concept of assisting the software engineer to create software from that of automatic program generation. Assistance leaves the creative decisions to the software engineer, but it tries to make available to the software engineer all of the information necessary to make the decisions. The assistance provided to the software engineer must be centered around the software engineer's current activities or focus of attention. In terms of computer based environments, the focus of attention is the terminal screen. Thus, the effective methods and environments that apply these methods, must organize information on the screen so that the software engineer can efficiently do the work of software construction.

Assistance can still be intelligent and do rule-based reasoning, but the ultimate decision is made by the software engineer. The best application for intelligent assistance is the summarization of pertinent information contained in other places so that the summary information can impact the ultimate decision. Intelligent assistance includes checking all of the rules for the application of a method to ensure that the software engineer does not violate any. For instance, SADT requires that any diagram can contain only 6 boxes. If the software engineer creates a seventh box then the environment should inform the engineer of the rule violation and suggest an alternative action.

To support many different methods, environments must either have the software engineering method hard coded into them or provide a feature for method specification. This is analogous to the relationship between files and databases. In databases the data model is used to create a representation of the data relationships, which is a separate process from actually entering, storing and retrieving data.

The same is true with environments. First the method (or methods) must be described in terms of the model. The environment can then be used to apply the method to an actual software design and implementation problem.

Beyond just representing the method, the environment must assist the software engineer in applying the method. This assistance can be done in several ways. The representation and flexibility of the model to represent many different types of methods is passive and limited to the model chosen to represent methods. In this case, assistance to the software engineer is provided merely by the power of the model to represent the methods and support them on a computer. Active assistance is provided when the environment can organize the method tasks for the software engineer. This organization relates not only to the information organized by the method, but also the placement of additional information or references to it for easy access. For example, links can be used by the environment to associate related pieces of information that may not be stored adjacently. Data flow diagrams are drawn at varying levels of detail. Links between the general level and the detailed level provide a fast means for an environment to access the information stored at the two levels. Further the available commands should be arranged such that the software engineer can select the next command based on the current context. This further implies that only those commands that are applicable based on the current context can be selected. The environment should provide an easy means for integrating external tools. Also the invocation of the tools should be done automatically based on the software engineer's context within the method. If the method has rules or procedures for operation, then the environment should provide facilities for encoding the procedures such that as the software engineer applies the method, the environment can apply the procedures based on previous input and modify existing information as well as synthesize new information.

2.5 ENVIRONMENTS TO SUPPORT MULTIPLE SOFTWARE ENGINEERING METHODS

In addition to the computer-based support requirements for software engineering methods, the following features are necessary to support software engineering methods within an environment:

o Mechanisms for relating information and structure between software engineering methods and

o Common storage representations.

Methods now exist to support each phase of the software life cycle. However, methods vary widely as to type (graphical as opposed to textual) and often are incompatible with each other. Further, a method used in one phase may not produce output suitable for use by a different method in the next phase. For instance, SADT used in the system design phase produces diagrams which cannot be directly translated into Jackson Method program designs.

Since the methods are not compatible phase to phase, there is a tendency by software engineers to only apply the method during the initial iteration of the software life cycle. If a requirements change is proposed during the system design phase, the likelihood is that its impact will be strictly on the system design and it will not be applied to the requirements analysis method to insure consistency and completeness with all of the other requirements. The greater the distance between the phases in which the error was found and the phase in which it was made, implies a greater loss in information. This tendency also undermines the documentation value of the method as a representation of the design, if the method is not being re-applied and updated as changes occur.

An environment should be capable of handling all the methods used during the software life cycle to create software. The ability to capture the information created during each phase of the software life cycle and apply it to the subsequent phases is necessary to provide complete computer based support to the software engineering process. An obvious solution to this problem is to create a new integrated method which can support each of the software life cycle phases. Not only is this a monumental task, but several of the existing methods are good for specific software engineering tasks and have been used successfully. Building an integrated method that is effective for all phases and consistent in use may be impossible considering the diverse activities involved involved in software specification, design and coding. An alternative solution is to provide an environment to uniformly support different methods in the software life cycle.

Further, providing a common interface to all the methods greatly strengthens the value of the methods to the overall software development process. In addition to sharing, the information can be propagated, which eliminates unnecessary copying of data and insures redundant information will be accurate. A common model for representing the methods also opens up the possibility of analyzing the results of method applications between software life cycle phases. For example, the ability to make sure that all requirements are met by software designs and implementation can be accomplished by linking requirements and their satisfying modules together between methods. A tool can then check and make sure that all requirements are paired and generate a report listing the pairings.

2.6 REQUIREMENTS FOR SOFTWARE ENGINEERING METHODS

This chapter has presented several software engineering methods to isolate these features of the methods that must be supported by a model for methods. Four basic requirements must be met to build a model for software engineering methods.

- o Represent the method structure,
- o Encapsulate the meaning of the structure,
- o Provide the capability for expressing the rules and procedures of the method use and
- o The model must be capable of being easily implemented on a computer so that computer based support can be supplied to these methods.

The structure of the method refers to the elements of the method, such as the boxes of the Jackson Method and the SADT Method. Also the model must be able to represent the connections between the elements, such as the lines of the Jackson Method or the arrows of the SADT Method. These elements are used to structure the software and the model must be flexible enough and robust enough to easily allow the expression of a variety of methods.

The meaning of the structure is the semantics of the arrangement of the method elements. For instance, the arrangement of subroutines into a hierarchical Call Structure means that if two modules are connected then one of the modules is above the other in the diagram. The model must be able to represent this meaning, too.

In addition to describing elements for representing software, most methods include rules which describe how the software is transformed into the elements of the method. For example, SADT requires that at most six boxes may be contained on any single page of an SADT Diagram. The model must accommodate the specification of such rules, so that the method can be correctly applied.

To facilitate the application of software engineering methods, the model must be amenable to easy implementation. This requirement extends computer support to existing methods that are currently unsupported or undersupported. It also provides the potential for computer support for new methods as they are created.

Further, representing the structure of software engineering methods, requires the following:

- o Chunking of concepts,
- o Representation of the connection of concepts (chunks) together as a directed graph. The connection should be capable of representing trees, hierarchies and networks,
- o Storage of text blocks,
- o Storage of attributes of the chunks,
- o Procedures to perform processing of stored information,
- o A Query Language to locate information based on the structure and content of the method and
- o Secondary Links to represent relations between method concepts different from the primary connections of the method.

Most software engineering methods attempt to provide either a compact notation for the software or an organized structure for the software. The elements of the method (notation or structure) allow for the concepts of the method to be "chunked" or aggregated. The model must provide a component to represent these "chunks".

Structure implies connection, so the model must allow the concepts to be connected and arranged into a meaningful structure embodied by the method. Numerous examples of these connections from software engineering methods have already been cited, most of which are arrows or arcs, but indentation in an outline is another way to organize or connect concepts in a text based method.

To support text based methods, the model must allow the inclusion of arbitrarily long sequences of text. At the other extreme, the model must allow for the definition and storage of attributes which describe the concepts of the method. For example, management methods, record dates, program sizes and percentages, all of which must be stored precisely for fast retrieval and manipulation.

Finally, a facility for building procedures to process the information represented by the method must be provided to build tools to translate the information to external sources or to other methods, or do local processing.

To effectively support the software engineer using a method expressed in the model, a query language is essential to locate information organized by the method. This requirement becomes more important as the size of the software represented by the method grows.

The following list of implementation requirements not only are characteristics of good software, but are necessary for a model to represent methods, if the resulting implementation is to be useful.

o Easy to use interface,

o Efficient and fast storage and retrieval of Entries and Units,

o Graphic views of Units and their Refinement Link structures,

o Robust and easy to use text editor and

o Flexible tool interface.

A good interface will allow the software engineer or application area specialist to use the model implementation easily with little training. An easy to use system will make the value of the model implementation greater.

A storage and retrieval mechanism can be used to store the elements of the TRIAD model. The mechanism can range from a B-tree scheme to a full featured database management system. However, the storage and retrieval mechanism must be efficient enough to accommodate method applications for large pieces of software. The response time must be fast enough to allow the software engineer to work without waiting for responses. A storage and retrieval mechanism should allow the implementation to support version control and multi-user access to a method use and method definition.

The efficient storage of the method structure and information contained within the structure, ensures that queries concerning the software in the method will be quickly answered. As with the interface, good response increases the likelihood that the model implementation will be used.

A package to provide graphic views of graphical methods such as Jackson, SADT or Dataflow Diagram is essential for a software engineer to use these methods with the model implementation. In addition to merely presenting a graphic view, the implementation should allow the user to manipulate the view directly which will result in the changes being recorded in the method representation.

The text editor is necessary for those methods which are largely textual, such as requirements or documentation methods. If the text editor is or resembles the standard one available on the host computer system, the user will be able to quickly begin entering and modifying the text in the method.

Finally a flexible tool interface is required to exploit existing tool support for some methods. The interface should contain primitives for extracting information from the method, as well as providing controlled invocation of the tool from the model implementation.

CHAPTER III

TRIAD MODEL DEFINITION

The preceding chapter provided the motivation for creating a model to represent software engineering methods. In addition, a brief description of the TRIAD model was presented in Chapter I. This chapter gives a precise formal definition of the two major components of the TRIAD model together with a description of the primary operations which the system provides for defining and using software engineering methods.

3.1 THE TRIAD MODEL

In order to support a variety of software engineering methods, the TRIAD model must have two components. The first component is a high level system which is used to specify particular methods such as the Call Structure method, the Jackson Method or the Dataflow Diagram Method. It is called the **Method Definition** Component and it allows the method definer to specify the names and general structures of the various general classes and categories of objects to be used in a particular method.

The second component of the model is a lower level system, the **Method Use** Component, which is used to manipulate individual usage instances of a particular method. It might, for example, be used to design a payroll program following the Jackson Method.

This division into two components is analogous to a similar division employed in databases. The Method Definition is analogous to the database schema, while Method Use is analogous to the storage and retrieval of data according to the schema.

3.1.1 THE TRIAD METHOD DEFINITION COMPONENT

A common feature of all software engineering methods is that they identify a small number of primary objects which are used to describe software. These objects would be the bubble and box of the Dataflow Diagram or the box of the Jackson Method. TRIAD uses the term **Unit Class** (UCl) to describe these objects. Figure 9 shows the formal definition of the Method Use portion of the TRIAD model. The Unit Classes are represented by the set in the upper right corner of the figure which is labeled UCl. Operations will be provided which allows a method definer, analogous to the database administrator, to identify the particular

Unit Classes which a method will use. One of the Unit Classes is designated as the **Initial Unit Class** to ensure that the network of Unit Classes is created properly. The Initial Unit Class (IUCL) is shown in the figure as a point contained within the set labelled, UCL.

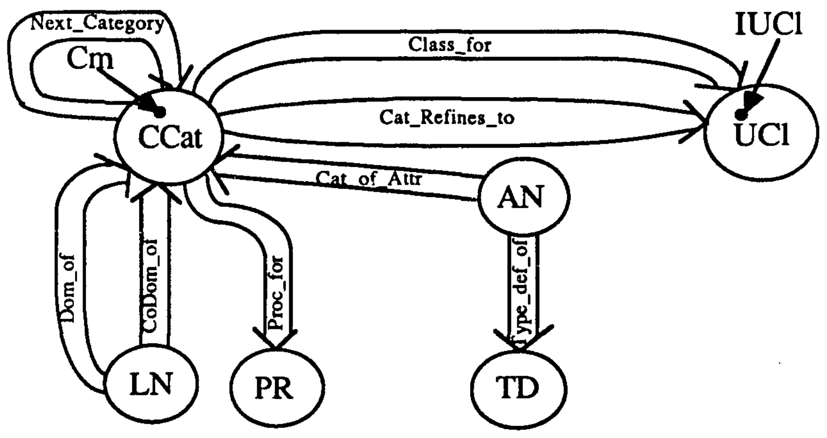


Figure 9. Method Definition Component of the TRIAD Model

Typically these Unit Classes will contain subcomponents such as the labels contained within the boxes of both the Dataflow Diagram and the Jackson methods. These labels describe the processing that the boxes represent. These subcomponents are identified in the TRIAD method definition system as **Component Categories** (CCat) and are represented in Figure 9 by the set labeled CCat in the upper left hand corner of the figure.

Each Component Category, y, belongs to a particular Unit Class, s. This is formalized in the TRIAD model using the Class_for function, which is shown in the figure as the arrow from the CCat set to the UCl set, and which is written as Class_for(y) = s. The model is constrained such that each Component Category, y, must map to some Unit Class, s. The Component Categories within each Unit Class are ordered by the sequence in which they are created. The Next_CCat function, which is shown in the figure as the circular arrow from the CCat set to itself, maintains the sequential order. That x is the next Component Category of a Component Category y is then denoted by Next_CCat(y) = x. The following constraint ensures that the Next_CCat function within a Unit Class points to only one Component Category and that the Component Categories do not precede each other. For all x and y in CCat, if Next_CCat(x) = Next_CCat(y) then x = y and Class_for(x) = Class_for(y) iff there exists an integer k such that Next_CCat ^{k} (x) = y or Next_CCat ^{k} (y) = x.

Each Unit Class has at least one Component Category which has the same name as the Unit Class and is used as a repository for information about the entire Unit Class rather than just a single Component Category. The notion of a Component Category is formalized by the function First_CCat which gives the first Component Category for each Unit Class. Note that if s is a Unit Class and $x = \text{First\_CCat}(s)$ then $\text{Class\_for}(x) = s$. In addition, to ensure that x is truly the first Component Category in the Unit Class, there is a constraint that for all y in CCat, if $\text{Class\_for}(y) = s$ then $\text{Next\_CCat}(y) \neq x$.

A **Method Cursor** labelled Cm, which is shown in the figure as a point within the CCat set, contains the current Component Category and provides a reference point for the method definer. The value of the cursor is changed by an operator which is used to navigate through the method definition.

The **Attribute Name** set defines the names of attributes which are used to hold descriptive information about the objects of the method. The Attribute Names are associated with a Component Category and in addition each Attribute Name has an associated **Type Definition** given by the Type_Def_of function shown in the figure as an arrow from the AN (Attribute Name) set to the TD (Type Definition) set.

For each AN, t, and TD, d, the function is formally defined as Type_Def_of(t) = d. Each Attribute Name, t, is associated with a Component Category, y, by the function Cat_of_Attr which is shown as the arrow from the AN set and to the CCat set in the figure and is formally defined as Cat_of_Attr(t) = y. This function associates the Attribute Names with the correct Component Categories.

The objects in software engineering methods are usually interconnected in various ways. In the case of the Jackson and Dataflow Diagram Methods, the boxes are connected by lines or arcs. In text based methods such as a documentation methods and requirements methods, the objects (descriptions) are typically connected according to their position in an outline thereby creating a hierarchy of objects. The TRIAD model represents these interconnections using the **Refinement Linkage.** This relation is from a Component Category, Y, to a Unit Class, S, and is shown in the figure as the fat arrow immediately below the Class_for arrow. The relation is formally defined as
Y Cat_Refines_to S.

In addition to the Refinement Linkage, other **Secondary Linkages** may be necessary to represent other relationships between the objects in the method. For example, both Jackson and the Dataflow Diagram break the processing into smaller pieces. At some point these pieces need to be combined into a program or into several modules, if the software is large. In the TRIAD model, Secondary linkages can be created between the first Component Category (which represents the entire Unit Class) to represent the grouping into modules. The Secondary Linkages are shown in the figure as the set labeled LN, Link Names. Between the LN and CCat are two arrows representing the domain, Dom_of function, and codomain, CoDom_of functions, for the link name. For a Link Name, n, and two distinct Component Categories, x and y, the functions are formally defined as Dom_of(n) = y and CoDom_of(n) = x.

In addition to the objects of a software engineering method, rules and procedures are provided to manipulate these objects according to the intent of the method. These rules are represented in the TRIAD model by procedures written in a programming language. The name and conditions under which these Procedures are to be invoked is associated with the Component Categories. The **Procedure References** are shown in the figure as the set PR in the lower left hand corner. The function Proc_for, shown in the figure as the arrow from the set CCat to the set PR, defines the Procedure Reference, p, for a Component Category, y, formally as Proc_for(y) = p.

Table 4 gives a formal definition of the Method Definition part of the TRIAD model.

Table 4. Formal Definition of the TRIAD Model

Method Definition Component

A method M is defined formally by a 17-tuple:

M=(CCat, UCl, IUCL, Next_Category, Cat_Refines_to, Class_for, First_Cat_of, LN, PR, AN, TD, Type_Def_of, Dom_of, CoDom_of, Proc_for, Cat_of_Attr, Cm)

1. CCat $\subseteq$ ch_strings is the set of Component Category names used by the method,
2. UCl $\subseteq$ ch_strings is the set of Unit Class names used by the method,
3. IUCL $\in$ UCl is the Initial Unit Class,
4. Next_Category: CCat -> CCat sequences the Component Categories for each Unit Class,
5. Cat_Refines_to $\subseteq$ CCat X UCl is a relationship which determines which Unit Classes a Component Category can refine to,
6. Class_for: CCat -> UCl determines the Unit Class each Component Category belongs to,
7. First_Cat_of: UCL -> CCat identifies the initial Component Category for each Unit Class,
8. LN $\subseteq$ ch_strings is the set of Link Names,
9. PR $\subseteq$ ch_strings is the set of Procedure References for the Component Categories of the method,

10. $AN \subseteq ch\_strings$ is the set of Attribute Names used in the method,
11. $TD \subseteq ch\_strings$ is the set of Type Definitions,
12. Type_Def_of: AN -> TD determines the type definition for each Attribute Name,
13. Dom_of: LN -> CCat determines the Domain Component Category for each Link Name,
14. CoDom_of: LN -> CCat determines the CoDomain Component Category for each Link Name,
15. Proc_for: CCat -> PR determines the Procedure Reference for each Component Category,
16. Cat_of_Attr: AN -> CCat tells which Component Category each Attribute is associated with,
17. $Cm \in CCat$ is the Cursor for the method and points to the Component Category currently being manipulated during method definition.

3.1.2 THE TRIAD METHOD USE COMPONENT

Technically, the method definition M provides a sort of template into which the particular software desired must be fitted. This is done by creating a variety of instances of the Unit Classes, Component Categories, etc. which were identified in the method definition. This process results in a set U of Unit Class instances which are called the **Units** of the particular method use. Similarly, the set C of Component Category instances will be called the **Components** of the method use and so forth.

As an example, if a software engineer is applying the Dataflow Diagram Method, then there will be a Unit Class "Processing Box" identified in the method definition. Each time he wishes to add a processing box to the software he is describing, he will ask the system to create a new Unit of the "Processing Box" class. If the software consisted of a source of data, two processing boxes and a data store, then four Unit Class instances would be created--one instance of the Unit Class "Source", one instance of Unit Class "Store" and two instances of Unit Class "Processing Box". An instance of a Unit Class automatically creates instances of all the Component Categories, Attribute Names and Link Names and the software engineer can use these to supply the details about the particular Unit.

The full TRIAD model is fairly complex and is depicted in Figure 10. The top part of the diagram, above the thick horizontal line, repeats the method definition given in Figure 9 while the lower portion of the diagram lays out the elements needed to describe a method usage. After a software engineering method has been translated into a TRIAD method using the Method Definition System, then the TRIAD Method Use System is available to create particular software documentation following the method defined.

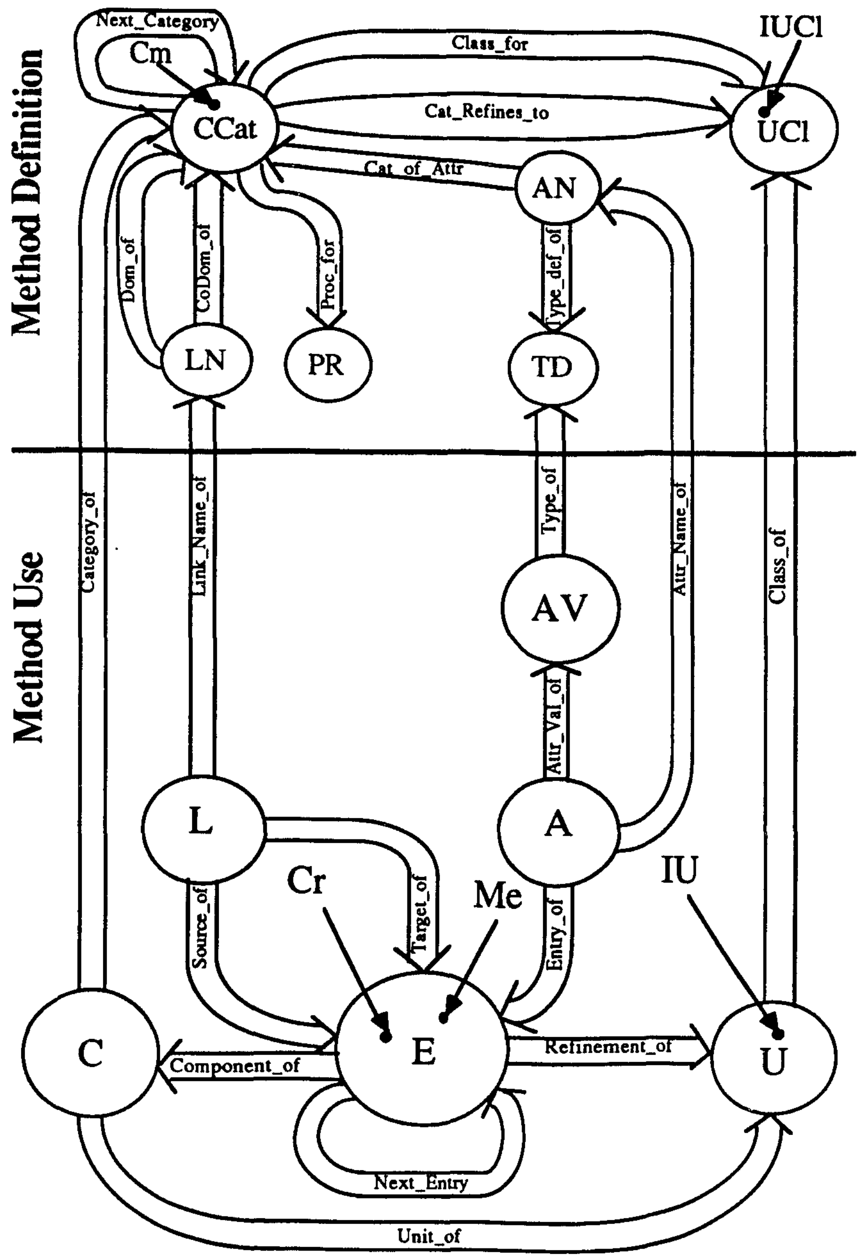


Figure 10. TRIAD Model

The use of a method is begun by the creating an **Initial Unit** which is in an instance of the Unit Class designated as the Initial Unit Class. For example, in the Dataflow Diagram Method the first symbol is a "source" for the program's data. The Initial Unit will, of course, be a member of the set of all **Units** which are represented in Figure 10 by the set U in the lower right corner. The Initial Unit Class (IUCL) is the point contained within the set U. The Class_of function shown in the figure as the arrow from the set U to the set UCl in the Method Definition portion of the figure records for each Unit created the Unit Class of which it is an instance. If Unit u is an instance of Unit Class S, then formally, Class_of(u) = s. A constraint on the model that the Initial Unit, IU, must be an instance of the Initial Unit Class is given formally as Class_of(IU) = IUCL.

Whenever a Unit is instantiated, new instances of all its Component Categories are created and added to the set C of **Components.** A "label" describing the contents of a "box" is an example of a Component in the Jackson Method or Dataflow Diagram Method. The Component represents the "label" for each box represented by a Unit instance. To record that Components, c, belong to particular Units, u, the Unit_of function is included in the model so that

Unit_of(c) = u. This function is shown in the figure as the arrow from the set C to the set U. Each Component c is created as an instance of a Component Category y and the model records this association by Category_of(c) = y. The Category_of function is the arrow on the left side of the figure from the set C to the set CCat. To maintain the consistency of the method use with the method definition, the category of the component must always belong to the same Unit Class as the Unit to which the Component belongs. For a Component, c, this constraint is formally defined as Class_for(Category_of(c)) = Class_of(Unit_of(c)).

Certain software engineering methods include components which actually contain an arbitrary number of entry items. For example, a project management method would allow an arbitrary number of programmer name entries in the "author" component of a "Module" Unit. In the TRIAD model then each Component Category may be replicated to permit sequences of method subcomponents. In the method use, **Entries** are created for each element in a sequence of subcomponents. At least one Entry is created for each Component. An example of the subcomponents in a method is the flow of data between symbols in the Dataflow Diagram. Data may go from one symbol to several other symbols. Each flow would be represented by a separate Entry. The Entries are shown as the large set in the middle at the bottom of the figure and is labelled E. The ownership of an Entry, e, by a Component, c, is denoted by the function Component_of, which is formally written as Component_of(e) = c. The function is shown as the arrow from the set E to the set C.

The Entries are in order in a component based on the order they are created. The function Next_Entry, which is shown in the figure as the circular arrow from the set E and to the set E, provides the means for navigating through the sequence of Entries in a Component. The function is formally defined for two adjacent Entries, e and f, as Next_Entry(e) = f. The following constraint ensures that the Next_Entry function within a Unit points to only one Entry and that only one entry precedes the other. For all e and f in E, if Next_Entry(e) = Next_Entry(f) then e = f and Component_of(Unit_of(e)) = Component_of(Unit_of(f)) iff for some integer k, Next_Entry (e) = f or Next_Entry (f) = e.

Attributes may be associated with each Entry according to the association of Attribute Names and Component Categories in the method use. For example, an Attribute Name was defined for the Jackson Method and Dataflow Diagram method to contain the symbol descriptions. The instance of the Attribute Name, the Attribute would contain the actual text describing the instance of the symbol represented by the Unit Class. The attributes are shown in the figure as the set labelled A in the middle at the right side. The function Entry_of, shown in the figure as an arrow from the set A to the set E, associates the Attribute, a, with an Entry, e, and is formally defined as Entry_of(a) = e. Shown in the figure as an arrow from the A set to the AN set in the method definition portion of the figure, the function Attr_Name_of establishes the correspondence between the Attributes, a, and the Attribute Names, t, and is formally defined as Attr_Name_of(a) = t. The values of the Attributes are contained in the set **Attribute Values** shown in the figure as the set labelled AV located above the set labelled A. The function Attr_Val_of shown in the figure as a label from the A set to the AV set, establishes the mapping from Attributes, a, to their Attribute Values, v, and is formally defined as Attr_Val_of(a) = v. Each of the Attribute Values must in turn have a type, which is defined by the Type_of function shown in the figure as the arrow from the set AV to the set TD in the method definition portion of the figure. This function is formally defined as for each Attribute Value, v, there is a Type Definition, d, such that Type_of(v) = t. To maintain the consistency between the method definition and the method use, two constraints are needed. The first constraint ensures that an Attribute, a, associated with an Entry in a Component has that Attribute Name related to the same Component Category that is mapped to the Component and is formally defined as Cat_of_Attr(Attr_Name_of(a)) = Category_of(Component_of(Entry_of(a))). The second constraint ensures that an Attribute, a, has an Attribute Value whose Type Definition is the same as that of the Attribute Name and is formally defined as Type_Def_of(Attr_Name_of(a)) = Type_of(Attr_Val_of(a)).

Refinement Links are shown in the figure as the arrow from the set E to the set U. The Refinement Link in the method use is an instance of the Refinement Linkage defined in the method definition. The Jackson and Dataflow Diagram Methods have arcs between the symbols which in the method definition are represented as Refinement Linkages from a Component Category to a Unit Class representing a symbol. In the method use, the Refinement Links are from an Entry, belonging to a Component in a Unit to another Unit, thus, representing the flow of control or data from one symbol to another. The function is shown in the figure as an arrow from the set E to the set U and is formally defined as for an Entry, e, there may exist a Unit, u, such that Refinement_of(e) = u.

In the same way that the contents of the Units must conform to the contents of the Unit Classes, the Refinement Links must be created in conformity to the Refinement Linkages in the method definition. For all e in E and for all u in U, if there is a refinement from e to u then the Component Category of the Component which contains the Entry, e, must be related to the Unit Class which the Unit, u, is an instance. This constraint is formally defined as if Refinement_of(e) = u, then Category_of(Component_of(e)) Cat_Refines_to Class_of(u). Units can not refine to themselves which is formally defined as Refinement_of(e) $\neq$ Unit_of(Component_of(u)).

The Is_Predecessor_of relation is defined to determine if two Units, u and v, are directly connected by way of a Refinement Link. If u is the predecessor of v then there is an Entry in the Unit u which refines to the unit v. This relation is formally defined as u Is_Predecessor_of v if and only if for some e in E, Refinement_of(e) = v and Unit_of(Component_of(e)) = u. To ensure that all units except the Initial Unit are refined to by at least one Entry, there must exist an integer k for all units such that through k applications of the Is_Predecessor_of relation, the Initial Unit can be reached. This constraint is formally stated as IU Is_Predecessor_of ^{k} u.

Secondary Links can also be created between Entries according to the Link Names defined in the method definition. These Secondary Links may be used to connect processing symbols together in either Jackson or Dataflow Diagram for example, into modules. An actual link instance is created from the Link Name according to the function Link_Name_of which is shown in the figure as the arrow from the set L, representing the Links, to the set LN, representing the Link Names. The function is formally defined for each Link, l, there must exist a Link Name, n, such that Link_Name_of(l) = n. The functions Source_of and Target_of provide the mappings of the Link, l, for the source and target entries of the link to the Entries, e,d, which are formally defined as Source_of(l) = e and Target_of(l) = d. These functions are shown in the figure as two arrows originating from the set L to the set E. To ensure that the links conform to the Link Name in the method definition which is mapped to from the Link, a constraint is placed on them such that the Source_of and Target_of functions must map to Entries whose Components are of the same Component Category as that specified by the Dom_of and Codom_of functions from the Link Name. This constraint is formally defined for all l in L,

Category_of(Component_of(Source_of(l))) =

Dom_of(Link_Name_of(l)) and

Category_of(Component_of(Target_of(l))) =

CoDom_of(Link_Name_of(l)). A further constraint is that only one Link with the same Link Name can have the same source and target entries. This constraint is formally defined for two Links, k and l, as

if Source_of(k) = Source_of(l) and

Link_Name_of(k) = Link_Name_of(l) or

if Target_of(k) = Target_of(l) and

Link_Name_of(k) = Link_Name_of(l) then k = l.

As in the method definition, a **Cursor**, Cr, is used to maintain a current position within a method use. The figure shows the method use cursor as a point within the set E. This cursor always points to an Entry and is used as the target entry for the method use operators requiring a target. When a source entry is also required by a operator, the **Mark Entry**, Me, represented by the other point within the set E is used.

The formal definition of the TRIAD model method use is given in Table 5. The constraints upon the TRIAD model are formally defined in Table 6 which follows the table containing the method use formal definition.

Table 5. Formal Definition of the TRIAD Model

Method Use Component

A Method Use S is defined for a method M is the 22-tuple:

S=(E, U, IU, Cr, Me, Next_Entry, Refinement_of, C, Unit_of, L, A, AV, Category_of, Class_of, Link_Name_of, Attr_Name_of, Attr_Val_of, Source_of, Target_of, Entry_of, Component_of)

1. E is the set of Entries,
2. U is the set of Units,
3. IU $\in$ U is the Initial Unit,
4. Cr is the method use cursor and points to the current entry being manipulated,
5. Me is the method use mark in the Entry set and points to an Entry,
6. Next_Entry: E -> E structures the entries for each component,
7. Refinement_of: E -> U determines the Unit to which each refinable Entry refines,
8. C is the set of Components,
9. Unit_of: C -> U determines the Unit each Component belongs to,
10. L is the set of Links,
11. A is the set of Attributes,
12. AV is the set of Attribute Values,

13. Category_of: C -> CCat maps the Components to Component Categories,
14. Class_of: U -> UCl maps the Units to Unit Classes,
15. Link_Name_of: L -> LN determines the Entry Link Name for each link,
16. Attr_Name_of: A -> AN determines the Attribute Name for each Attribute,
17. Attr_Val_of: A -> AV determines the Attribute Values for each Attribute,
18. Type_of: AV -> TD determines which Attribute Value is of which Type Definition,
19. Source_of: L -> E determines the Source Entry for each Link,
20. Target_of: L -> E determines the Target Entry for each Link,
21. Entry_of: A -> E determines the Attribute associated with each Entry and
22. Component_of: E -> C determines the Entries in each Component,

Table 6. TRIAD Model Constraints

Let the function First_CCat be defined by

First_CCat(s : UCl) = x <u>and</u> Class_for(x) = s <u>and</u>

For all y $\in$ CCat if Class_for(y) = s then Next_CCat(y) $\neq$ e

<u>Lemma</u> First_CCat is a total function

Next_Category Constraint:

For all x,y: CCat if Next_CCat(x) = Next_CCat(y) then x=y and Class_for(x)=Class_for(y) iff there exist a k: N such that Next_CCat (x) = y or Next_CCat ^{k} (y) = x

Next_Entry Constraint:

For all d,e: E if Next_Entry(d) = Next_Entry(e) then d=e and Component_of(Unit_of(d)) = Component_of(Unit_of(e)) iff there exists a k: N such that Next_Entry ^{k} (d) = e or Next_Entry ^{k} (e) = d

Component Category Contents Constraint:

For all y $\in$ CCat there exists an s $\in$ UCl such that Class_for(y) = s

Initial Unit Constraints:

There exists an IU $\in$ U and an s $\in$ IUCL such that Class_of(IU) = s

Let the relation Is_Predecessor_of $\subseteq$ U X U be defined by u Is_Predecessor_of v iff there exists an e $\in$ E such that Refinement_of(e) = v and Unit_of(Component_of(e)) = u

Connectivity Constraint:

For all $u \in U$ there exists a k such that

IU Is_Predecessor_of^{k} u

Unit Contents Constraint:

For all $c \in C$,

Class_for(Category_of(c)) = Class_of(Unit_of(c))

Refinement Constraint:

For all $e \in E$ and $u \in U$, if Refinement_of(e) = u then

Category_of(Component_of(e)) Cat_Refines_to

Class_of(u) and

Refinement_of(e) $\neq$ Unit_of(Component_of(e))

Component Constraints:

For all $e \in E$, there exists a $c \in C$,

such that Component(e) = c

Attributes Constraints:

For all $a \in A$, Cat_of_Attr(Attr_Name_of(a)) =

Category_of(Component_of(Entry_of(a))) and

Type_Def_of(Attr_Name_of(a)) =

Type_of(Attr_Val_of(a))

Links

For all $l \in L$,

Category_of(Component_of(Source_of(l))) =

Dom_of(Link_Name_of(l)) and

Category_of(Component_of(Target_of(l))) =

```
    CoDom_of(Link_Name_ of(l)) and
For j,k ∈ L, if (Source_of(j) = Source_of(k) and
  Link_Name_of(j) = Link_Name_of(k)) or
  (Target_of(j) = Target_of(k) and
  Link_Name_of(j) = Link_Name_of(k)) then j=k
```

3.2 TRIAD MODEL OPERATORS

The TRIAD model operators are divided into two groups corresponding to the two components of the TRIAD model. The method definition operators allow the method definer to create and modify the sets comprising the method definition. Table 7 lists the operators (in pairs where appropriate) and a brief description of the operator's function. The target of an operator is assumed to be the position of the method cursor within the Component Category set. The word "current" when applied to the Component Category refers to the category currently pointed to by the method cursor.

Table 7. Method Definition Operators

Create/Delete Method creates/deletes an entire method definition

Add/Delete Unit adds/deletes a Unit Class. The Add_Unit operator also creates the first Component Category in the class giving it the same name as the class

Add_Category adds a new non-refinable Component Category following the current Component Category.

Add_Refinable_Category adds a new refinable Component Category following the current Component Category.

Delete_Category deletes the current Component Category.

Add_Type_Definition adds a new type to the set of type definitions.

Add/Delete Attribute adds/deletes an Attribute Name. The Add_Attribute operator also tells which type definition belongs to the Attribute Name .

Add/Delete Link Name adds/deletes a link name from the current Component Category.

Add/Delete PC Reference adds/deletes a PC reference from the current Component Category.

Next/Previous Category moves the method cursor to the next/previous Component Category if the cursor is not already pointing to the last/first Component Category in the Unit Class.

First_CCat positions the method cursor at the first
Component Category within the named Unit Class.

The method use operators manipulate the sets specified during method definition. Many of these operators assign values to the names defined previously. As with the method definition operators, a cursor is used to specify the default target entry for the operators. For those operators requiring a source as well as a target, an additional cursor called the Mark Entry is provided. The word "current" applied to an entry refers to the entry currently being pointed to by the cursor.

Table 8 contains the names of the method use operators and a brief description of their function.

Table 8. Method Use Operators

Use/Delete Method	uses/deletes a method use.
Create_Unit	creates a copy of Unit Class. An Entry and Component is created for each Component Category within the Unit Class.
Mark_Entry	sets the additional cursor to point to the Entry pointed to by the cursor.
Refine	creates a Refinement Link from the current Entry to the Unit pointed to by the Mark Entry.
Delete_Unit	deletes the Unit which is the Unit of the current Entry, provided the current Entry is the first Component of the Unit. Also the Unit must not have any Secondary Links or additional Refinement Links connected to it.
Replicate_Entry	creates another Entry of the same Category following the current Entry in the same Component as the current Entry.
Delete_Replicate	deletes the current Entry, provided it is a replicate within a Component and that it is not the last replicate.
Change_Attr_Val_of	changes the value of the specified Attribute associated with the current Entry to the new specified value. The value must be of the same type as that defined for the Attribute name.

Change_Link changes the specified link (source or target) to be the current Entry.

Follow_Link follows the specified link which has either the current Entry as its source or target and sets the cursor to the named link's target or source.

Move_Entry moves the current Entry to follow the Entry pointed to by the Mark Entry. Both Components must be of the same Category.

Next/Prev Component sets the cursor to the first/last Entry in the next Component within the same Unit provided the cursor is not already pointing to the last/first Component within the Unit.

Next/Prev Entry sets the cursor to the next/previous Entry within the current Component provided the cursor is not already pointing to the last/first Entry in the Component.

Visit_Refinement sets the cursor to the first Entry within the Unit which the current Entry refines to provided the current Entry is refinable and has been refined to a Unit.

3.2.1 TRIAD MODEL METHOD DEFINITION OPERATORS

Table 9 gives the formal definition for each TRIAD method definition operator. Each operator is presented with its name, parameters, pre and post conditions (require and ensure) and a description of the operator. All parameters are assumed to be constant. The number sign (#) preceding a name indicates that the name represents the old value as opposed to the current value. The $\ddagger$ symbol represents the undefined value for a function and $\emptyset$ is the empty set.

Table 9. Formal Definition of the

TRIAD Method Definition Operators

Operation Create_Method(Start_Unit_Name : ch_string)

Require $CCat=\emptyset$ and $UCl=\emptyset$ and $AN=\emptyset$ and $PR=\emptyset$ and $IUCL=\emptyset$

Ensure IUCL = Start_Unit_Name and

UCl = {IUCL} and

CCat = {IUCL} and

Class_for(IUCL) = IUCL and

Cm = IUCL

Description The Create Method operator begins the definition of a new method. It creates the first Unit, named Start_Unit_Name and places this name in the Initial Unit Class (IUCL). The first Component Category (CCat) is also created with name Start_Unit_Name.

Operation Delete_Method;

Require $CCat\neq\emptyset$ and $UCl\neq\emptyset$ and $AN\neq\emptyset$ and $PR\neq\emptyset$.

Ensure $CCat=\emptyset$ and $UCl=\emptyset$ and $AN=\emptyset$ and $PR=\emptyset$.

Description The Delete Method operator deletes the current method.

```
Operation Add_Unit(Unit_Name :  ch_string)
  Require Unit_Name ∉ #UCl
  Ensure UCl = #UCl U {Unit_Name} and
         CCat = #CCat U {Unit_Name} and
         Class_for(Unit_Name) = Unit_Name and
         Cm = Unit_Name;
  Description The Add Unit operator creates a new Unit Class
      with name Unit_Name and the first Component Category
      in the unit is created with the same name, also.  The
      method cursor, Cm, is set to point to the first entry
      for the entire unit.
Operation Delete_Unit
  Require #Cm = First_CCat(Class_for(#Cm))
  Ensure UCL = #UCL - {Class_for(Cm)} and
         For all y: CCat, [ CCat = #CCat - {y} and
           For all n: LN
             Dom_of(n) = ⊥ iff #Dom_of(n) = y and
             CoDom_of(n) = ⊥ iff #CoDom_of(n) = y
           For all t: AN,
             Cat_of_Attr(t) = y iff #Cat_of_Attr(t) = y and
             y ≠ #Cm) ]
         iff Class_for(y) = Class_for(#Cm) and
         Cm = IUCL
  Description The Delete Unit operator deletes the Unit
```

Class of the is the unit for the Component Category pointed to by the method cursor, Cm. The cursor must be on the first category of the class. All components which are members of the deleted unit are also deleted.

Operation Add_Category(CCat_Name : ch_string)

Require CCat_Name $\notin$ #CCat

Ensure CCat = #CCat U {CCat_Name} and

Class_for(CCat_Name) = Class_for(#Cm) and

Next_Category(#Cm) = CCat_Name and

Cm = CCat_Name

Description The Add Category operator adds a non-refinable Component Category with name CCat_Name following the CCat pointed to by the method cursor, Cm.

Operation Add_Refinable_Category(CCat_Name, Unit_Name : ch_string)

Require CCat_Name $\notin$ #CCat

Ensure CCat = #CCat U {CCat_Name} and

Class_for(CCat_Name) = Class_for(#Cm) and

Next_Category(#Cm) = CCat_Name and

Cm = CCat_Name and

CCat_Name Refines_to Unit_Name

Description The Add Refinable Category operator adds a refinable Component Category with name CCat_Name following the CCat pointed to by the method cursor, Cm. The new category refines to the Unit Class named Unit_Name,

<u>Operation</u> Delete_Category

<u>Require</u> cm $\in$ #CCat <u>and</u>

#Cm $\neq$ First_CCat(Class_for(#Cm))

<u>Ensure</u> CCat = #CCat - {#Cm} <u>and</u>

For all y: CCat,

[Cm = y iff [#Cm = #Next_Category(y) <u>and</u>

$$\text{Next\_Category}(y) = \begin{cases} \text{\#Next\_Category(\#Cm)} \\ \quad \text{iff \#Next\_Category}(y) = \text{\#Cm} \\ \text{\#Next\_Category}(y) \text{ otherwise } \underline{\text{and}} \end{cases}$$

For all n: LN

$$\text{Dom\_of}(n) = \begin{cases} \perp \text{ if \#Dom\_of}(n) = \text{\#Cm} \\ \text{\#Dom\_of}(n) \text{ otherwise } \underline{\text{and}} \end{cases}$$

$$\text{CoDom\_of}(n) = \begin{cases} \perp \text{ if \#CoDom\_of}(n) = \text{\#Cm} \\ \text{\#CoDom\_of}(n) \text{ otherwise } \underline{\text{and}} \end{cases}$$

For all t: AN,

Cat_of_Attr(t) = y

iff [#Cat_of_Attr(t) = y <u>and</u>

y $\neq$ #Cm]]

iff Category_of(Component_of(#Cm) = y

<u>Description</u> The Delete Category operator removes the Component Category pointed to by the method cursor, Cm, as long as the cursor is not pointing to the first

Category in the Class.

Operation Add_Type_Definition(Type_Name : ch_string)

Require Type_Name $\notin$ TD

Ensure TD = #TD U {Type_Name}

Description The Add Type Definition operator adds a new type definition to the TD set.

Operation Add_Attribute(Attr_Name, Type_Name : ch_string)

Require Attr_Name $\notin$ AN and
Type_Name $\in$ TD and
Cat_of_Attr(Attr_Name) $\neq$ #Cm

Ensure AN = #AN U {Attr_Name} and
Cat_of_Attr(Attr_Name) = #Cm and
Type_Def_of(Attr_Name) = Type_Name

Description The Add Attribute operator adds the Attribute Name named, Attr_Name and of type, Type_Name to the Component Category pointed to by the method cursor, Cm.

Operation Delete_Attribute(Attr_Name : ch_string)

Require Attr_Name $\in$ AN

Ensure AN = #AN - {Attr_Name}

Description The Delete Attribute operator removes the Attribute Name Attr_Name from the Entry_Category pointed to by the method cursor, Cm.

Operation Add_Link_Name(Link_Name, CCat_Dom, CCat_CoDom : ch_string)

Require Link_Name $\notin$ #LN

Ensure LN = #LN U {Link_Name} and

CCat_Dom = Dom_of(Link_Name) and

CCat_CoDom = CoDom_of(Link_Name).

Description The Add Link Name operator adds the Link Name Link_Name to the Component Category pointed to by the method cursor, Cm with domain and codomain specified by CCat_Dom and CCat_CoDom, respectively.

Operation Delete_Link_Name(Link_Name : ch_string)

Require Link_Name $\in$ #LN

Ensure LN = #LN - {Link_Name}

Description The Delete Link Name operator deletes the Link Name named Link_Name of the Entry Category pointed to by the method cursor, Cm.

Operation Add_PC_Reference(PC_Name : ch_string)

Require PC_Name $\notin$ #PR

Ensure PR = #PR U {PC_Name} and

PC_Name = Proc_for(#Cm).

Description The Add Procedures Reference operator adds a Procedure name to the Procedures Reference (PR) set. The reference is attached to the Entry Category pointed to be the method cursor, Cm.

Operation Delete_PC_Reference(PC_Name : ch_string)

Require PC_Name $\in$ #PR and

Proc_for(#Cm) = PC_Name

Ensure PR = #PR − {PC_Name} and

Proc_for(#Cm) = $\perp$

Description The Delete Procedures Reference operator deletes the Procedures Reference named PC_Name which is associated with the Component Category pointed to by the method cursor, Cm.

Operation Next_Category

Require #Next_Category $\neq \perp$

Ensure Cm = #Next_Category(#Cm).

Description The Next Component Category operator sets the method cursor to point at the next entry category in the Unit Class by applying the Next_Category function.

Operation Previous_Category

Require #Cm $\neq$ Class_for(#Cm)

Ensure For all y: CCat, Cm = y iff #Next_Category(y)=#Cm

Description The Previous Category operator sets the method cursor to point at the previous entry category in the Unit Class by applying the inverse of the Next_Category function. This operation is not performed if the cursor is at the first Component Category of the Unit Class.

The method structure is defined using the above operators. The software engineer defining the method uses the Add/Delete Unit Class and Add/Delete Component Category operators to define the structure of the method. The Next_Category function allows the navigation through the method definition. Attributes and links may be added at any time. The method cursor is used as the default for any of the operators requiring a target.

When a Component Category is defined, it can be specified as refinable using the Add_Refinable_Entry and therefore one or more Unit Classes must be named to which the Category refines to. If more than one Unit Class is specified for the Cat_Refines_to relation then this Component Category can refine to any one of the Unit Classes named, but only one. Therefore a selection or alternate feature is allowed for refinement. Also Attribute Names can be created and associated with either the Component Category pointed to by the method cursor or the Unit Class which the Component Category pointed to by the method cursor is contained in.

If the method is specified top down (the first unit defined has references to undefined units) then it is necessary to keep track of all Unit Class names so that the uniqueness of the names can be preserved. Maintaining the Unit Class name uniqueness implies not allowing a unit to be defined with the same name as an existing Unit Class. Also when deleting a Unit Class, the specified Unit Class must be defined. Further, when a Unit Class is deleted, all references to it must be marked as undefined. Before a method can be used and Unit instances created, all references to undefined Unit Classes must be satisfied by either defining the Unit Class or by removing the reference.

3.2.2 TRIAD MODEL METHOD USE OPERATORS

Table 10 gives the formal definition of the method use operators.

Table 10. Formal Definition of the Method Use Operators

```
Operation Use_Method
  Require $L = \emptyset$ and $A = \emptyset$ and $C = \emptyset$ and $E = \emptyset$ and $U = \emptyset$ and
          $IU = \perp$ and $IUCL \neq \perp$
  Ensure $U = IU$ and
         Class_of(IU) = IUCL and
         For all y: CCat,
           There exists a $c \in C$ such that $\{c\} = C - \#C$ and
           Category_of(c) = y and
           Unit_of(c) = u and
           There exists an $e \in E$ such that
             [ $\{e\} = E - \#E$ and
             Component_of(e) = c and
             Cr = e iff Category_of(Component_of(e)) =
               First_CCat(Class_of(Unit_of(Component_of(e)))
               and
             For all l: L, there exists an $l \in L$ such that
               [ $\{l\} = L - \#L$ and
               Source_of(l) = e
                 iff Category_of(Component_of(e)) =
                   Dom_of(Link_Name_of(l)) and
               Target_of(l) = e
                 iff Category_of(Component_of(e)) =
                   CoDom_of(Link_Name_of(l)) ]
```

```
                    For all a: A, [ {a} = A - #A and
                      Entry_of(a) = e and
                      There exists a t: AN such that
                        Attr_Name_of(a) = t iff
                        Cat_of_Attr(t) = y ]
                      iff
                      Category_of(Component_of(Entry_of(a)))
                      = y]
                    iff Class_for(y) = IUCL and
                    For all e: E, Cr = e iff
                      Category_of(Component_of(e)) =
                        First_CCat(Class_of(IUCL))
```

<u>Description</u> The Use Method operator begins the use of a method. The Initial Unit (IU) which is of type Initial Unit Class is created. The cursor is set to point to the first entry in the unit.

<u>Operation</u> Delete_Method_Use

<u>Require</u> IU $\neq \perp$

<u>Ensure</u> L = $\emptyset$ <u>and</u> A = $\emptyset$ <u>and</u> C = $\emptyset$ <u>and</u> E = $\emptyset$ <u>and</u> U = $\emptyset$

<u>Description</u> The Delete Method Use operator deletes the current method use.

```
Operation Create_Unit
  Require There exists an s: UCL such that
          Category_of(Component_of(#Cr)) Cat_Refines_to s
  Ensure These exists a u ∈ U such that {u} = U - #U and
         Refinement_of(#Cr) = u and
         Class_of(u) = Category_of(Component_of(#Cr))
           Cat_Refines_to and
         For all y: CCat,
           There exists a c ∈ C such that {c} = C - #C and
           Category_of(c) = y and
           Unit_of(c) = u and
           There exists an e ∈ E such that
             [ {e} = E - #E and
             Component_of(e) = c and
             Cr = e iff Category_of(Component_of(e)) =
               First_CCat(Class_of(Component_of(Unit_of(e)))
               and
            For all l: L, there exists an l ∈ L such that
              [ {l} = L - #L and
              Source_of(l) = e
                iff Category_of(Component_of(e)) =
                  Dom_of(Link_Name_of(l)) and
              Target_of(l) = e
                iff Category_of(Component_of(e)) =
                CoDom_of(Link_Name_of(l)) ]
          For all a: A, [ (a) = A - #A and
            Entry_of(a) = e and
            There exists a t: AN such that
              Attr_Name_of(a) = t iff Cat_of_Attr(t) = y ]
            iff Category_of(Component_of(Entry_of(a))) = y]
          iff Class_for(y) = Class_of(u) and
          For all e: E, Cr = e iff
            Category_of(Component_of(e)) =
              First_CCat(Class_of(u))
  Description The Create Unit operator creates a Unit whose
      class is determined by the value of the Entry pointed
      to by the Cursor, Cr.  The cursor must be on a
      refinable entry.
Operation Mark_Entry
  Require #Cr ≠ ⊥
  Ensure Me = #Cr
  Description The Mark Entry operator marks the current
      Entry pointed to by the Entry Cursor, Cr.
Operation Refine
  Require Refinement_of(#Cr) = ⊥ and
          Me ≠ ⊥
  Ensure Refinement_of(#Cr) = Unit_of(Component_of(Me)) and
         There exists an e: E, such that [ Cr = e
```

```
          Category_of(Component_of(e)) =
            First_CCat_(Unit_of(Component_of(Me))
  Description The Refine operator creates a Refinement Link
      from a non-refined refinable entry pointed to by the
      cursor, Cr, to the Unit of the entry pointed to by the
      Mark Entry, Me, which was previously set by the
      Mark_Entry operator.
Operation Delete_Unit
  Require Refinement_of(#Cr) ≠ ⊥ and
           For all e: E,
             [ Refinement_of(e) = Refinement_of(#Cr)
               iff e = #Cr and
           For all l: L and e: E,
             [ Source_of(l) = e iff e = #Cr and
             [ Target_of(l) = e iff e = #Cr and
               Source_of(l) = ⊥ ] ]
           iff Unit_of(Component_of(#Cr)) =
             Unit_of(component_of(e))
  Ensure U = #U - {Refinement_of(#Cr)} and
          Refinement_of(#Cr) = ⊥ and
          [ For all e: E,
            [E = E - {e} and
            For all a: A, A = #A - {a}
              iff Entry_of(a) = e and
            For all l: L, L = #L - {l}
              iff Source_of(l) = e or Target_of(l) = e ]
          iff Unit_of(Component_of(e)) = Refinement_of(#Cr) ]
  Description The Delete Unit operator deletes the Unit
      which is the refinement of the Entry pointed to by the
      Entry Cursor, Cr.
Operation Replicate_Entry
  Require Category_of(Component_of(#Cr)) ≠
            First_CCat(Class_of(Unit_of(Component_of(#Cr))))
  Ensure There exists an e: E such that {e} = E - #E and
            Component_of(e) = Component_of(#Cr) and
            Cr = e and
            Next_Entry(e) = Next_Entry(#Cr) and
            Next_Entry(#Cr) = e
  Description The Replicate operator creates a new entry,
      following the one pointed to by the cursor, Cr, of the
      same category and in the same unit and component.
Operation Delete_Replicate
  Require Category_of(Component_of(#Cr)) ≠
            First_CCat(Class_of(Unit_of(Component_of(#Cr))))
  Ensure E = #E - {#Cr} and
          For all a: A,
            A = #A - {a} iff Entry_of(a) = #Cr and
          For all l: L, L = #L - {l}
```

iff [Source_of(l) = #Cr or Target_of(l) = #Cr]

For all e: E,

$$\text{Next\_Entry}(e) = \begin{cases} \#\text{Next\_Entry}(\#\text{Cr}) \text{ iff } \\ \quad \#\text{Next\_Entry}(e) = \#\text{Cr} \\ \#\text{Next\_Entry}(e) \text{ otherwise} \end{cases}$$

Description The Delete Replicate operator removes the entry if it is not refined to a unit, in the component which the cursor is currently pointing to,

Operation Change_Attr_Val_of(Attr_Name : ch_string, Attr_Val_of : AV)

Require There exists an a: A, such that Attr_Name_of(a) =
Attr_Name and
Entry_of(a) = #Cr and
Cat_of_Attr(Attr_Name) =
Category_of(Component_of(#Cr))

Ensure There exists an a: A, such that Attr_Val_of(a) =
Attr_Val_of and
AV = #AV U {Attr_Val_of} and
Type_of(Attr_Val_of) =
Type_Def_of(Attr_Name_of(a)))

Description The Change Attribute Value operator changes the value of the Attribute whose name is Attr_Name and is associated with the Entry pointed to by the cursor, Cr,

```
Operation Change_Link(Link_Name : ch_string)
  Require [ Dom_of(Link_Name_of(Link_Name)) =
              Category_of(Component_of(#Cr)) or
            CoDom_of(Link_Name_of(Link_Name)) =
              Category_of(Component_of(#Cr)) ] and
            [ Source_of(Link_Name) = #Cr or
              Target_of(Link_Name) = #Cr ]
  Ensure Source_of(Link_Name) = Me
             iff Target_of(Link_Name) = #Cr and
           Target_of(Link_Name) = Me
             iff Source_of(Link_Name) = #Cr
  Description The Change Link operator sets the source or
      target (whichever points to the current Entry) of the
      link named Link_Name to the new entry pointed by the
      Entry Mark, Me.
Operation Follow_Link(Link_Name : ch_string)
  Require Source_of(Link_Name) = #Cr or
          Target_of(Link_Name) = #Cr
  Ensure Cr = Target_of(Link_Name)
             iff Source_of(Link_Name) = #Cr and
           Cr = Source_of(Link_Name)
             iff Target_of(Link_Name) = #Cr
  Description The Follow Link operator moves the cursor to
      the entry pointed to by the source or target
```

(whichever points to the current Entry) of the link named Link_Name.

Operation Move_Entry

Require Me $\neq \perp$ *and*

Category_of(Component_of(Me)) =

Category_of(Component_of(#Cr))

Ensure Component_of(Me) = Component_of(#Cr) *and*

For all e: E,

$$\text{Next\_Entry(e)} = \begin{cases} \text{Me iff \#Next\_Entry(e)} = \text{\#Next\_Entry(\#Cr)} \\ \text{Next\_Entry(Me) iff \#Next\_Entry(e)} = \text{\#Next\_Entry(Me)} \\ \text{\#Next\_Entry(\#Cr) iff \#Next\_Entry(e)} = \text{Me} \\ \text{\#Next\_Entry(e) otherwise} \end{cases}$$

Description The Move Entry operator moves the marked Entry from its current place to a place following the Entry pointed to by the cursor Cr,

Operation Next_Component

Require #Cr $\neq \perp$ *and*

Next_Category(Category_of(Component_of(#Cr))) $\neq \perp$

Ensure There exists an e: E, such that Cr = e iff

Next_Category(Category_of(Component_of(#Cr)) =

Category_of(Component_of(e)) *and*

For all f: E, Next_Entry(f) $\neq$ e

<u>Description</u> The Next Component operator sets the cursor to the next Component in the Unit unless the cursor is pointing to an Entry of the first Component.

<u>Operation</u> Prev_Component

<u>Require</u> #Cr $\neq \perp$ <u>and</u> First_CCat(Class_of(Unit_of(Component_of(#Cr)))) $\neq$ Category_of(Component_of(#Cr))

<u>Ensure</u> There exists an e: E such that, Cr = e iff Next_Category(Category_of(Component_of(e))) = Category_of(Component_of(#Cr)) <u>and</u> For all f: E, Next_Entry(f) $\neq$ e

<u>Description</u> The Previous Component operator sets the cursor to the first Entry in the preceding Component in the Unit if the cursor is not already set to the first Component in the Unit.

<u>Operation</u> Next_Entry

<u>Require</u> Next_Entry#Cr) $\neq \perp$

<u>Ensure</u> Cr = Next_Entry(#Cr)

<u>Description</u> The Next Entry operator sets the cursor to the next entry in the component,

<u>Operation</u> Prev_Entry

<u>Require</u> #Cr $\neq \perp$ <u>and</u>

There exists an e: E, such that Next_Entry(e) = #Cr

<u>Ensure</u> For all e: E, Cr = e iff Next_Entry(e) = #Cr

<u>Description</u> The Previous Entry operator sets the cursor to the preceding Entry in the Component if the cursor is not already set to the first Entry.

<u>Operation</u> Visit_Refinement

<u>Require</u> Refinement_of(#Cr) $\neq \perp$

<u>Ensure</u> There exists an e: E, such that Cr = e iff

First_CCat(Class_of(Refinement_of(#Cr))) =

Category_of(Component_of(e))

<u>Description</u> The Visit Refinement operator sets the cursor to point to the first Entry in the Unit which is the refinement of the Entry which the cursor is currently pointing at,

The first time a method is used for a new piece of software, a Unit from the Initial Unit Class (the Initial Unit) is created. The cursor is set to the first Entry within the Unit. From the Initial Unit, all of the other Units are created. A Unit can only be created from an Entry in a Component with a valid reference to a Unit Class, which is a refinable Component Category. The process of creating a Unit also creates Entries and Components for all Component Categories belonging to the unit class.

Deletion of a Unit is accomplished by reversing the process of creating instances. Units are deleted by positioning the cursor to the Entry that refines to the Unit to be deleted. The Refinement Link is removed and the Entry is returned to its original state (before it was refined). If the removed Refinement Link was the only one to the Unit and there are no Secondary Links between Entries in the Unit to be deleted and Entries in other Units, then the Unit is destroyed. If another Refinement Link refers to the Unit to be deleted, then only the link from the Entry from which the deletion was initiated is deleted. The link from the referenced Unit to the Entry is removed and the Unit is left intact. If the Unit to be deleted has no additional Refinement Links from other Entries, but does have Secondary Links referencing it, then the deletion is not permitted until the method user explicitly removes the Secondary Links. The same sequence of events is applied to every Unit that is referenced (either by Refinement Links or by Secondary Links) by a Unit to be deleted. The delete operator must not ruin the integrity of the Refinement Links by removing a Unit that is refined to by another Entry.

The Replicate Entry operator creates another Entry in a Component. The Delete_Replicate operator removes a replicated Entry from the Component providing the Entry is not currently refined to another Unit.

The Change Attribute Value operator allows the software engineer to maintain the values of the Attributes. This operator implies the use of a text editor to change the long strings of text that may be stored in an Attribute. The actual form of the text editor is left to the implementor, but the editor should have the operators to add, delete, change and search text, in addition to operators for moving through the text based on characters, words sentences and paragraphs.

The Change Link operator allows the source or target of Secondary Links to be changed. Secondary Links are Entry to Entry links, except when the links are between the first Entries of Units, then the links are essentially Unit to Unit links. These links provide the method definer with the means to connect entire Units together with a single link type.

During method application it is possible for the user to move entries from one position in a Component to another position in the same or different Components. Both Components (source and target) must be of the same Category.

If the Entry being moved has references to other Units by way of links (either Refinement or Secondary), the references are left intact, thus, this operation has the effect of altering the network of the Units. This operation is essentially a combined Delete Entry and Replicate Entry operator, because the links are removed from the source Unit and moved to the target Entry.

A query package provides a general purpose capability for searching the structure and contents of the TRIAD model. It is not necessarily a single operator, but several. It should search for Unit, Entry and Attribute Names as well as the Attribute Values. This query capability should be as robust as those found with database management systems.

The use of a method often suggests changes in the method definition. Some changes are subtle and only involve a name change for a unit or entry, while others may create new units and delete existing ones. The process of changing a method that has already been applied is called "Tuning".

3.3 TUNING A METHOD

Tuning can be of two types--local or global. Local tuning involves changing the structure and not the content of a Unit. Local tuning is restricted to changing the names of Entries, adding or deleting Attributes and adding or deleting Secondary Links. The changes are only applicable to the Unit being tuned. All other Units of the same Unit Class are unaffected, hence the reason for the name local tuning. Additional Component Categories can be added during local tuning, however, changes in the structure of the Unit Class often means a weakness in the software engineering method definition. Structural changes are best made as global tuning actions to keep the Units consistent with the Unit Classes.

Global tuning involves changing the Unit Classes in the same manner as when the method was first defined. However, since the method has already been partially applied, all changes must applied to each Unit of the same Class to keep future Units consistent with existing Units. The same checks that were made for the Delete Unit operator are also made during global tuning when a refinable Entry is removed or a Unit Class is deleted.

Although global tuning by default affects the entire collection of Units, it is sometimes desirable to globally tune only a subset of the Units. Global tuning of a subset causes a consistency problem if any Unit Class has Units included and excluded from the subset. After the global tuning of such a Unit Class is complete and when the next instance of the Unit is made the new globally tuned version is used. The result of this tuning is the elimination of the excluded Unit Class. This problem is overcome by changing all Entries refining to the unit to specify more than one Unit Class to refine to. Then the Entry can be refined to either the original Unit (excluded from the subset) or the new Unit changed through global tuning (included in the subset). In the Call Structure example, the Unit class is "MODULE". After a Call Structure is defined using this software engineering method, suppose that the program represented is greatly expanded and new modules coded in a different programming language are added. In this case the method designer wants to change the "MODULE" unit to add new Entries specific to the programming language used to implement the module. Rather than creating one Unit Class with language specific Entries, different Unit Classes are created for each language and the Entry refining to the "MODULE" Unit must refine to a particular type of "MODULE" such as CMOD, FORTMOD, PLIMOD etc.

3.4 TRIAD PROCEDURES

Additional features of the TRIAD model can be expanded from the basics defined above. Most of these features are achieved through the implementation. One such feature, Procedures, is very basic to the use of the TRIAD model for representing software engineering methods. The TRIAD model supports the definition of the references to Procedures, but the actual construction of the Procedures is left to the Method Designer. They are built from whatever languages and compilers are available in the implementation of the model. A Procedure is written in a programming language. The Procedure is used by the method designer to express the procedural aspects of using a method. For example, rules for the use of a method can be implemented using a Procedure. Procedures can also be used as tool interfaces and to implement extended commands. Operators are provided for the Procedure to manipulate and process the information stored in the methods defined using the TRIAD Model. Procedures are invoked based on access to an Entry where the Procedures reference is attached. When an Entry is accessed, the Procedure invocation rules, which are stored as Attributes of the Entry, are checked and only those Procedures satisfying a two component rule get invoked. The first component of the rule is the invoking agent, which is either the user (by way of a direct command), an extended command or another Procedure. In the latter two cases, the name of the extended command or Procedure must match the invoking agent name. The second component of the invoking criteria is the entry/unit status. The following 5 status are possible:

- Create,
- Delete,
- Enter,
- Exit and
- Modify.

These states correspond to user access actions, thus one Procedure can be invoked when the user enters (applies the Next function to change the cursor) an Entry and another one when the user exits the Entry.

For instance, the display of the entry may cause a Procedure to be invoked which will dynamically count the lines of code contained in an adjacent Entry containing the program source code. In this example, the invocation criteria is the display of the form. Other criteria can include removing the Entry from display, modifying the entry text or access of the Entry by a tool.

Procedures use implementation provided operators to do processing in the Units, but are prohibited from altering the structure of the Units (delete Units or changing links). This restriction eliminates the possibility of deadlock situations caused by indirect invocation of one Procedure by another Procedure.

3.5 USER VIEW OF THE TRIAD MODEL

Although the definition of the TRIAD model is in terms of sets, functions and relations, the software engineer using the TRIAD model sees it differently. Although the user interface is dependent on the implementation of the model, a rudimentary description here of the user view of the model will facilitate the discussion of the application of the model to software engineering methods. Figure 11 shows the basic structure of the user view of the TRIAD model, which is a Unit Class containing a refinable and non-refinable component categories.

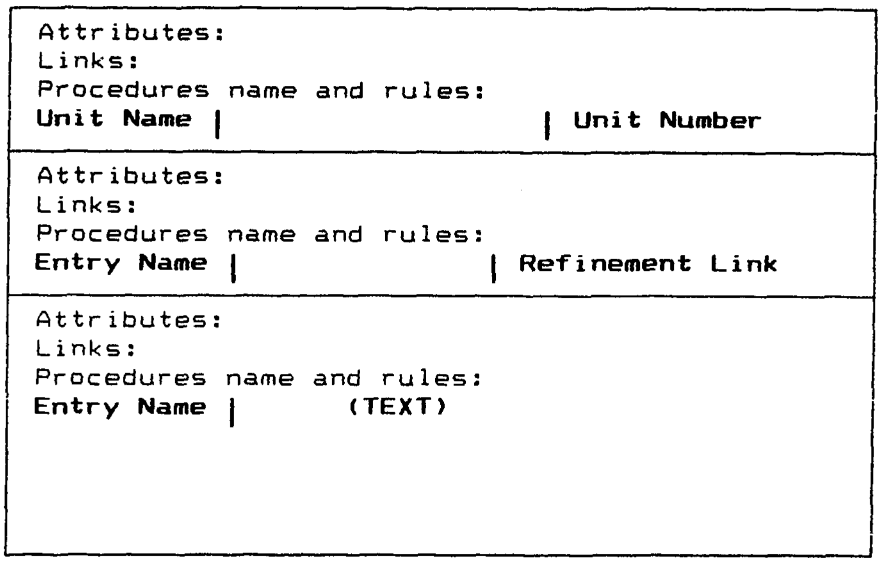


Figure 11. User View of the TRIAD Model

The visible parts of the unit are the box surrounding the Unit, the vertical lines separating the Entries and the Entry Names or tags (printed in dark type). Located above each Entry in the Unit are the Attributes. The Secondary Links and Procedure References are special types of Attributes, but are show here to emphasize their value to method definition and use.

Note that the user view parallels the model in that the groups of Attributes are clustered together into Component Categories represented by the boxes surrounding them. All the Categories are surrounded by a frame which represents the Unit Class.

3.6 USING THE TRIAD MODEL TO REPRESENT A METHOD

The Call Structure example in Figure 6 from Chapter II is used to illustrate the TRIAD model. First the Call Structure method will be defined using the method definition elements of the TRIAD model. Next, the method definition will be used to apply the method to the name and address file maintenance example.

To represent the Call Structure of a group of subroutines or modules, a Unit Class called "MODULE" is created. "MODULE" has an Attribute associated with it which contains the name of the module. Two Component Categories are contained in the "MODULE" Unit Class. The first is the Component Category "PROGRAMMER" which records the name of the programmer responsible for the module. "PARAMETERS" is the next Component Category. It contains the names and type of the parameters required for the module which are contained in attributes associated with the entry. This Component Category is capable of being replicated, which allows more than one parameter to be specified for each module. The next Component Category is "SOURCE", for the

source code of the module. An Attribute which is of type text, contains the actual source code. Following the "SOURCE" Component Category is the "CALLS" category. "CALLS" is refinable to the "MODULE" Unit Class. An Attribute containing the name of the module being called is associated with "CALLS". Since this example has only one Unit Class, "MODULE" is the Initial Unit Class, also.
The outline below summarizes this example method definition.

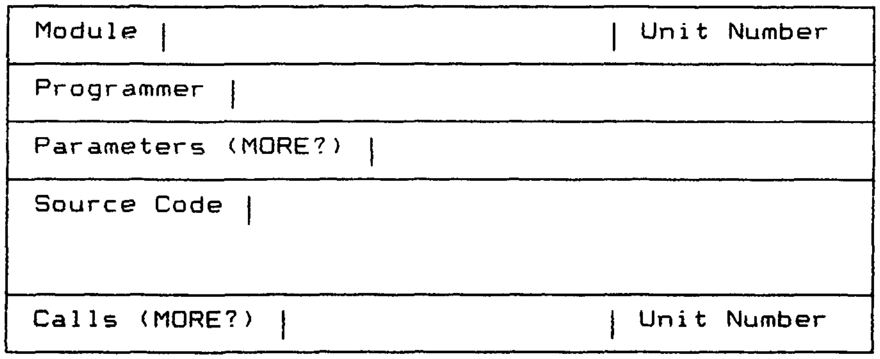
```
    Unit Class: MODULE
    Attribute: (name_of_module;ch_strings)
        Component Category: PROGRAMMER
        Attribute: (name;ch_string)

        Component Category: PARAMETERS
        Attribute: (replicable;integer)
        Attribute: (parameter_name;ch_string)
        Attribute: (parameter_type;ch_string)

        Component Category: SOURCE
        Attribute: (source_code;text)

        Component Category: CALLS (refines_to;MODULE)
        Attribute: (name_of_called_module;ch_string)
```

The user view of the Call Structure method is shown in Figure 12.



Module |	| Unit Number
Programmer |	
Parameters (MORE?) |	
Source Code |	
Calls (MORE?) |	| Unit Number

Figure 12. Module unit

Applying this method to the Call Structure example in Chapter II produces the network of units shown in Figure 13.

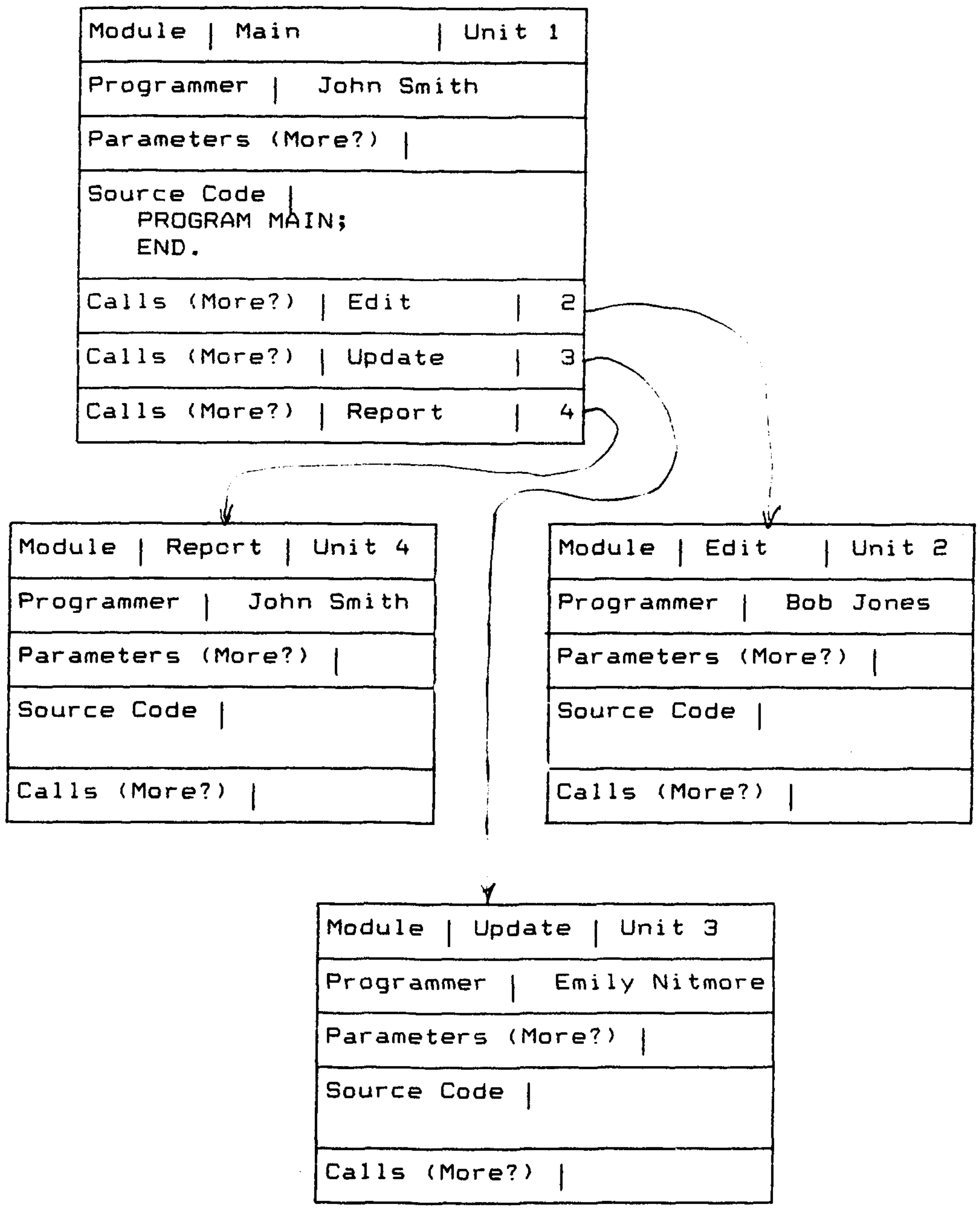


Figure 13. Instantiated TRIAD Model Units

CHAPTER IV

ALTERNATIVE MODELS

The development of a model to represent software engineering methods draws from several areas of computer science research. Some methods have a rigid structure and share many properties in common with programming languages. In addition, those methods that are primarily textual require a sophisticated text editor to apply and maintain the text contained in the method. Both of these features indicate that grammars and the related syntax directed editors are appropriate to represent some software engineering methods.

The assistance a software engineer receives from a computer based method is largely due to the storage and retrieval of the information organized by the method. Data models are useful for representing methods and databases are extremely beneficial for the actual storage and retrieval of the information.

In the future, artificial intelligence (AI) research will contribute much to the techniques for applying expert programmer knowledge to software engineering problems. The research done in AI on knowledge representation is essential to ultimately represent expert programmer knowledge. Until expert programming knowledge can be captured and used, research on knowledge representation can be practically applied to assist the software engineer in developing software.

Although the TRIAD Model was constructed by examining the research contributions of these three areas, not one of the three provides a single model strong enough on its own to support methods description and application. However, the combination of elements from these three areas embodied in the TRIAD model does provide a superior model.

4.1 GRAMMAR FORM

Soni, Kuo and McKnight have developed the Grammar Form Model for the representation of methods based on attribute grammars [SONI83, KUO83, MCKN85]. The method is specified by writing production rules for a grammar which will accept the method. An attribute grammar is a quadruple

$G=(G_0, A_G, A, \text{sem})$ where

o $G_0=(V,\Theta,P,\sigma)$ is a grammar,

o A_G is a specification of attributes,

o A is an attribute associator for G and A_G and

o sem is a semantic function association for productions in G such that sem(p) is a valid collection of semantic functions for p in P.

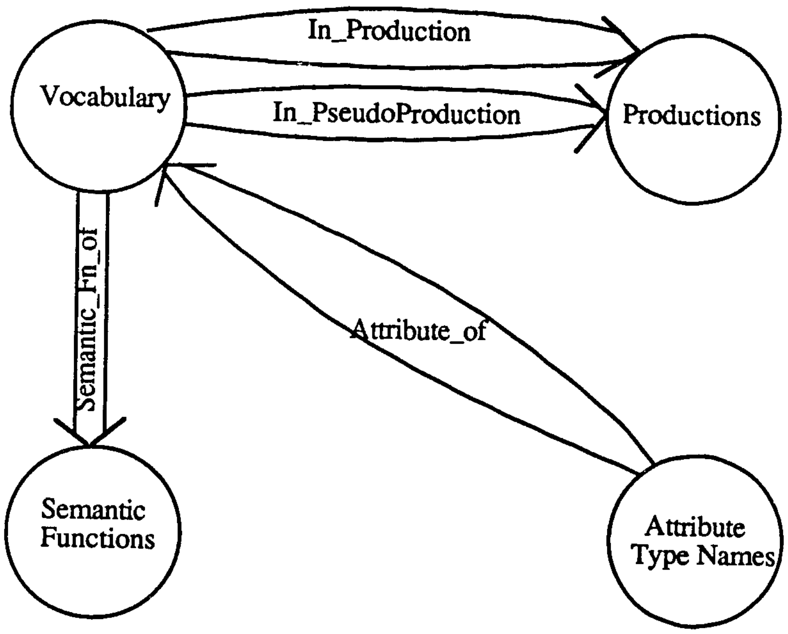


Figure 14. Grammar Form Model

The method definition portion of the Grammar Form Model is shown in Figure 14 The three circles represent (left to right) the Vocabulary (G), the Productions (P), the set of semantic functions and the specification of Attributes (A). The relation between the Attributes and the Vocabulary is the attribute associator (A_G). The function Semantic_Function_of maps the semantic functions to the symbols. The relation In_Production_of relates the symbols in the Vocabulary (V) to Productions (P). The relation In_PseudoProduction_of relates some of the Non-terminal symbols in the Vocabulary (V) to pseudo productions which define the form view of the method. These productions are of the form S->S'.

The method is defined in the Grammar Form Model by describing a grammar. The Component Categories correspond to the symbols. The method definer writes productions to represent the structure of the symbols. For example, the call structure example in Chapter III can be represented in the Grammar Form Model as follows:

V = {Programmer Parameters Source Module}

P = {Module -> Programmer, Parameters, Source, Module}

In this example Module on the right hand side of the production represents the "CALLS" Entry. Module has a dual role, it is both a left hand side symbol and a right hand side symbol. As a right hand side symbol it represents a symbol belonging to the production and as a left hand side symbol it represents a refinement to a new production. This ambiguity is resolved by introducing a pseudo production, Module' -> Module. Now the productions for the call structure example are:

Module -> Programmer Parameters Source Module'

Module' -> Module.

The attributes and semantic functions are equivalent in both models and will not be expanded in this example. The method use is not represented in Figure 14 because the Grammar Form Model defines a grammar, which is merely used to generate correct sequences in the "language" (method definition). The use of the method is therefore the application of the grammar generated by the method definition.

Two major deficiencies of the Grammar Form Model as opposed to the TRIAD Model are readily apparent. The first is that the Grammar Form Model does not explicitly support links between productions and symbols as the TRIAD Model does with the entry category links. However, links can be simulated in the Grammar Form Model by storing the path from one symbol to another symbol as an attribute. This technique requires additional storage (the sum of the path lengths to the common parent production) and additional computation to locate the ends of the links. The TRIAD Model stores the location of the link source and target and can access the entries directly in one operation. Although this deficiency can be overcome through a clever implementation, the method definer has a more difficult time conceptualizing Secondary Links with the Grammar Form model then with the TRIAD model. The difficulty in conceptualizing may affect the quality or the range of software engineering methods that may be represented.

Secondly, the Grammar Form Model produces a tree representation of the method and therefore cannot represent graphical methods such as Dataflow Diagrams and the call structure method. On the other hand, the TRIAD Model's Refinement Linkages can represent directed graphs and the Secondary Links achieve network representations.

Table 11 compares the method definition of the TRIAD Model to the Grammar Form Model.

Table 11. Comparison of TRIAD and Grammar Form Models

TRIAD Model	Grammar Form Model
Component Categories (CCat)	Vocabulary (V)
Attribute Names (AN)	Specification of Attributes (A)
Unit Classes	Productions (P)
Next_CCat	Next_Symbol

Table 11 (continued)

Cat_Refines_to	In_PseudoProduction_of
Unit_for	In_Production_of
Is_Attr_of_Cat, Is_Attr_of_Class	Attribute Associator (A_a)

The specification of a method is a different process using the Grammar Form Model than that of the TRIAD Model. The method definer is specifying a grammar and must define the sets constituting the grammar. McKnight describes the following steps in method specification [MCKN85]:

o Define Symbol Set - The vocabulary and start symbol,

o Define Production Rule Set - the relations between the symbols,

o Define Attribute Set - the attributes associated with the symbols,

o Define Action Set - the semantic functions associated with the productions,

o Define Blank form Set - the mapping to the method user's view of the method,

o Compile Method Description - Check the consistency of the sets defined above and create a grammar to use the defined method.

The following operators are available in the Grammar Form Model to define a method:

- o Create_A_Method(Method_name, Start_Symbol : ch_string) creates a new method called Method_Name with the start symbol named Start_Symbol,
- o Delete_A_Method(Method_Name : ch_string) deletes the method named Method_Name,
- o Add_A_Symbol(Symbol_Name : ch_string) adds the symbol named Symbol_Name to the symbol set,
- o Delete_A_Symbol(Symbol_Name : ch_string) removes the symbol named Symbol_Name from the symbol set,
- o Does_The_Symbol_Exist(Symbol_Name : ch_string) checks the symbol set to see if the symbol named Symbol_Name exists,
- o Add_A_Production(Production_Name : ch_string) adds the production named Production_Name to the production set,
- o Delete_A_Production(Production_Name : ch_string) removes the production named Production_Name from the production set,
- o Add_To_A_Form(Form_Name, Production_Name : ch_string) adds the production named Production_Name to the form named Form_Name,
- o Delete_From_A_Form(Form_Name, Production_Name : ch_string) deletes the production named Production_Name from the form named Form_Name,

- o Add_An_Attribute(Symbol_Name, Attribute_Name, Attribute_Type : ch_string) adds the attribute of type Attribute_Type and named Attribute_Name to the symbol named Symbol_Name,
- o Delete_An_Attribute(Symbol_Name, Attribute_Name : ch_string) removes the attribute named Attribute_Name from the symbol named Symbol_Name,
- o Does_The_Attribute_Exist(Symbol_Name, Attribute_Name : ch_string) checks the symbol named Symbol_Name to see if the attribute named Attribute_Name exists,
- o Add_A_Semantic_Function(Function_Name, Production_Name : ch_string) adds the semantic function named Function_Name to the production named Production_Name and
- o Delete_A_Semantic_Function(Function_Name, Production_Name : ch_string) removes the semantic function named Function_Name from the production named Production_Name.

The method use operators for the Grammar Form Model are defined as follows:

- o Create_Form_Tree(Tree_Name) creates a new form tree with name, Tree_Name,
- o Starting_Form_Tree(Form_Name) starts the form tree with the blank form named Form_Name,

- o Delete_Form_Tree(Tree_Name) removes the form tree named Tree_Name,
- o Refine(Entry_Name,Form_Name) refines the entry named Entry_Name to the form named Form_Name,
- o Choice(Entry_Name) select the entry named Entry_Name from a set of alternate entries (productions),
- o More(Entry_Name,n) make n copies of the entry named Entry_Name,
- o Delete_Entry(Entry_Name) delete the entry named Entry_Name,
- o Next_BlankEntry(Entry_Name) find the next unfilled entry named Entry_Name,
- o Next_Entry(Entry_Name) find the next entry named Entry_Name,
- o Next_Unrefined_Entry(Entry_Name) find the next unrefined entry named Entry_Name,
- o Visit_Form(Form_Number) visits the form with number Form_Number and
- o Child_Form(Entry_Name,Form_Number) visits the form with number Form_Number which is refined to from entry named Entry_Name,
- o Parent_Form(Form_Number) visits the parent form with number Form_Number and
- o Search_for_... includes several special operators which search for occurrences of symbols, attributes and text occurring within entries and forms.

Table 12 compares the method definition operators of the TRIAD Model to the Grammar Form Model.

Table 12. Comparison of Method Definition Operators

TRIAD Model	Grammar Form Model
Create_Method	Create_A_Method
Delete_Method	Delete_A_Method
Add_Unit	Add_To_A_Form
Delete_Unit	Delete_From_A_Form
Add_Entry and Add_Refinable_Entry	Add_A_Symbol,Add_A_Production
Delete_Entry	Delete_A_Symbol,Delete_A_Production
Query	Search_for
Add_Attribute	Add_An_Attribute, Add_A_Semantic_Function
Delete_Attribute	Delete_An_Attribute, Delete_A_Semantic_Function
Add_Link_Name	
Delete_Link_Name	
Next_CCat and Previous_CCat	Does_The_Symbol_Exist

Although the Grammar Form Model and the TRIAD Model method definition operators appear to be very similar, there are several major differences. The first major difference is the lack of secondary links in the Grammar Form Model. The organization of the symbols is by way of the parse tree and access to all symbols is done by navigating through the tree.

The second major difference is the process of defining the method. The Grammar Form Model requires the method definer to define the set of symbols and then the set of productions which structure the symbols into a method. The fifth row in Table 12 has two operators for the TRIAD Model and two for the Grammar Form Model. However, the two TRIAD operators differentiate between the two types of Component Categories, refinable and non-refinable, but perform the same task that of adding an Component Category to a unit class. On the other hand, the two Grammar Form Model operators perform separate operations. The first adds a symbol to the symbol set and the second adds a production to the production set. Thus the Grammar Form Model requires two operators to define an entry in the model, which is done with a single operation (choice of two operators based on the type of entry) in the TRIAD Model.

Tying the Grammar Form Model productions to the form view is a third major difference between the two models. The TRIAD Model has a uniform representation for both the method definition and use, while the Grammar Form Model uses a grammar to represent the method and a form based interface to use the method. The Add_To_Form operator associates a production with a blank form name. All productions are tied to forms on the basis of the derivation tree. The form assignment is made for a production and all productions derived from the production with the form specified are tied to the same form until another form assignment is found. Although the TRIAD Model Add_Unit is some what equivalent to the Add_To_A_Form operator of the Grammar Form Model, the Add_Unit operator is used to create a Unit Class. All subsequently defined Component Categories are members of that Unit Class which is referenced by the cursor. The Grammar Form Model uses the Add_To_A_Form operator after all of the productions are defined.

Finally the method definer has operators in the Grammar Form Model to search the sets of symbols, attributes and productions, which are unnecessary in the TRIAD Model. When a method is defined in the TRIAD Model, the method definer has all of the information needed to define the Unit Classes and the Component Categories. In the Grammar Form Model, the method definer has to build the sets, independently or constantly change between the sets if an incremental approach is used. Even after the symbols are defined and the productions written, the mapping to the form view is yet another disjoint operation.

Table 13 compares the method use operators of the TRIAD Model to the Grammar Form Model.

Table 13. Comparison of Method Use Operators

TRIAD Model	Grammar Form Model
Use_Method	Create_Form_Tree, Starting_Form_Tree
Delete_Method_Use	Delete_Form_Tree
Create_Unit and Refine	Refine, Choice
Delete_Unit	Delete_Form
Replicate	More
Delete_Replicate	Delete_Entry
Change_Attr_Value	
Create_Link, Delete_Link and Follow_Link	
Mark_Entry and Move_Entry	
Next_Entry	Next_Blank_Entry, Next_Organizer and Next_Unrefined_Organizer
Visit_Unit	Visit_Form
Visit_Child_Unit	Child_Form
Visit_Parent_Unit	Parent_Form
Query	Search_For

The method use operators between the two models are very similar. Again, the absence of secondary links in the Grammar Form Model means that the link operators are present for the TRIAD Model only. The Grammar Form Model has more specific navigation operators then the TRIAD Model. However, this is only a convenience factor and the same more specific operators could be constructed for the TRIAD Model by combining the Next_Entry functions and the query operator.

The TRIAD Model because of its ability to represent graphs, has two separate operators for refinement. The Create_Unit operator creates a new Unit from the refinable Entry and also completes the Refinement Link between the Entry and the new Unit. The Refine operator is used to refine a refinable Entry to a Unit that already exists. In this case, the operator completes the link from the Entry to the specified Unit.

The following is a list of the major advantages of the TRIAD Model over the Grammar Form Model for providing a precise model which best represents software engineering methods.

o Representation

- Directed graphs can be represented using the Refinement Linkages in the TRIAD Model whereas only trees can be directly represented in the Grammar Form Model
- The TRIAD Model supports Secondary Links from Entry

to Entry thereby allowing the capability to represent networks. The Grammar Form Model does not have secondary links.

- The Grammar Form Model is best for representing language based methods while the TRIAD Model is appropriate for language type methods and other, less structured methods.
- The TRIAD Model has a uniform view of method definition and use, while the Grammar Form Model uses a grammar for method definition and a form
- The TRIAD Model is more natural for expressing methods than the grammar approach. The software engineering can express the method definition in a representation as close to the method as possible. No translation to a grammar is necessary.

o The TRIAD Model uses a direct manipulation, incremental approach to specifying and using a method, while the Grammar Form Model requires the method to be defined as a grammar, in disjoint sets.

o The use of grammars to specify a method is different from the classic use of grammars as recognizers of sentences in a language. The grammar form is used as a generator of grammars. The generated grammar being the method specification.

4.2 DATABASE MODELS

The definition of a software engineering environment has three components, an editor, interface and storage facility. The obvious comparison of a software engineering environment to a database is natural. Classical database model implementations--hierarchy, network and relational--are oriented towards transaction based processing of fixed format fields. Little support for large blocks of unparsed text is provided, particularly for editing or searching [KENT79]. Therefore, the availability of a database implementation to use directly without modification for method support is not possible. The hierarchical model, like the grammar form is unsuitable for method specification because of the difficulty in representing directed graphs. Although the relational model contains the expressive power to represent any structure including directed graphs, it is difficult to capture the semantics of the method stored in the relations. The creation of data dictionaries and the Entity-Relationship and Semantic Data Model are solutions to the need to represent not only the structure of the data, but the meaning of the structure.

The semantics of data refers to the meaning of the structure. Databases have a model for structuring data, a query language for retrieving data from the structure and a procedural language for writing extended commands and programs to access the database. Each one of these features is separate. The software engineering environment needs processing embedded within the structure of the data (method). By embedding the processing within the method, processing can be defined for classes of data, which will be available for all instances of the class when the method is used. Processing which is invoked based on data access, enables the environment to offer assistance to the software engineer applying the method. This assistance would have to be provided for each method by the person defining the method. This is a different approach then that of writing a single database program to control the user's interactions with the database. It is a local approach that attaches the procedure references to the data, causing the interaction to be triggered by access.

Although the relational data model could be used to build a software engineering method representation, the TRIAD Model captures the essence of software engineering methods structure as atomic features. Further the TRIAD Model provides support for incorporating the knowledge to apply the methods with the structure, something the classical models do not provide.

The Entity-Relationship Model (E-R) proposed by Chen attempts to capture the meaning of data by naming the relations and the entities [CHEN76]. The model is intended to be built upon the relational model and used by the Database Administrator at a cognitive level for describing the data. The E-R model is naturally intended to be a general model for the universe of database applications.

The goal in creating the TRIAD Model is to build a specialized model capable of capturing the distinct software engineering method support requirements. Although the E-R model, like the relational model, has the expressive power to represent methods, it lacks the method specific features of the TRIAD Model.

The specification of the relationships in the E-R Model are also present in the Secondary Links of the TRIAD Model. The Secondary Links are named at method definition time by the method specifier. The primary links (refinement) are already specified as ownership links.

Several new data models have been proposed [BROD84, TSIC82]. These new models allow the database administrator to create new data types that contain predefined restrictions, attributes, processing functions and relationships to other types. The classical data models merely organized the data without explicitly allowing the database user to use the schema other than to specify the record and field names. In fact, the creation of a separate data dictionary by several commercial database implementations to help the user organize and remember the many record and field names, illustrates this void in the classical data models.

A final problem exists with most database implementations. The data definition is analogous to the model definition in the TRIAD Model, however; the method definition can be interactively changed by tuning the method. Most database implementations require the data definition to be recompiled and the data translated to the new structure. Both of these operations are usually done in batch mode. To use a database as an implementation vehicle, it must support dynamic changes to the data definition and be capable of allowing embedded procedure references with the data.

The most promising (and most complicated) new data model is the semantic data model (SDM) which combines the schema and data into a network [HAMM81]. Although the SDM is appropriate for method definition, it is complicated and difficult to use. SDM is a much more general model for data representation, while the TRIAD model is focused on representing and supporting methods.

4.3 KNOWLEDGE REPRESENTATION FRAMES

Some software engineering methods are a first attempt at applying artificial intelligence (AI) techniques to software construction. Although most methods do not automatically produce programs as an expert system would, they are attempts at recording representations and the knowledge of expert programmers in terms of the techniques used to produce software. It is natural then that knowledge representation ideas should be applied to software engineering methods. A prominent knowledge representation scheme is the AI frame [MINS75]. The frame was proposed as a model for use in computer vision, but since has been expanded and applied to the representation of knowledge for deduction as well as recognition. Basically a frame represents a stereotype of a concept. It has fixed items which are always present and slots for specific information--instances of the concept. Thus, the frame serves as a combined schema and storage cell. Demons are also associated with the frame and are used to represent procedural knowledge. Further, frames, may be connected together into a network of frames, thereby representing a body of knowledge.

Although there are many similarities between AI frames and the TRIAD Unit, there are several important differences. The first major difference is one of purpose. AI frames are used to not only represent knowledge, but also to support the recognition of the concepts represented. The first application of AI frames was to vision and natural language recognition. Their use was extended to not only recognize, but also show the path through the frames, thereby, demonstrating the reasoning used to recognize a concept.

The use of demons is different from that of TRIAD Procedures. The demons are used in the AI frame as recognizers and fire automatically once a concept is presented for recognition. The Procedures attached to the TRIAD Entries are invoked in a more orderly fashion, often as the result of the user moving the cursor on the display terminal. For example, the AI frame demons may all fire and try to recognize a concept, whereas the TRIAD Procedure may only be invoked if the software designer displays the representation for a module on the terminal screen.

To support current software engineering methods, less automatic reasoning is required. The TRIAD Unit is used more as a storage entity, letting the software engineer do the reasoning. Thus, the structure of the TRIAD Unit Class is borrowed from AI, but the application is different.

CHAPTER V

SUPPORT FEATURES OF THE TRIAD MODEL FOR SOFTWARE ENGINEERING METHODS

The TRIAD Model was designed to support the definition and use of software engineering methods. This chapter describes how the requirements for a model for software engineering methods, which were described in Chapter II, are met by the TRIAD model. The last section describes how the features of the TRIAD Model support multiple software engineering methods.

Four basic requirements were given in Chapter II for a model to represent software engineering methods. They are:

o Represent the method structure,

o Encapsulate the meaning of the structure,

o Provide the capability for expressing the rules and procedures of the method use and

o The model must be capable of being easily implemented on a computer so that computer based support can be supplied to these methods.

The ability of the TRIAD model to represent the structure of software engineering methods was informally and formally given in Chapter III. The next section describes those features along with some extensions derived from the model which better support the method structure.

The next requirement, that of capturing the meaning of the method structure, is accomplished in the TRIAD model by the Method Definition Component. The Method Definition is not only a flexible device for expressing software engineering methods using a general model, but it is also retained through Method Use as a reference and recording of the method definition. The structural features of the model entities are named, which includes the Component Categories, Unit Classes, Attributes, Links and Procedures. These names can be used both by the method definer and method user to gain insight into the meaning of the method structure. The names can be used in the query language, to extract relationships between method objects. Further, the Procedures can be created to analyze the method structure and make the meaning clear. For example, the Call Structure Diagrams (and other hierarchical methods) use the position of the boxes within the diagram to not only represent calls, but also scope and successor and predecessor relationships. Procedures can be written to capture the meaning of the position generally in the method definition, then at method use, the Procedure can show the relationship of the actual instances of the software within the Call Structure hierarchy.

The third general requirement, that of a facility for expressing the rules of the method is captured in the Procedures. Finally, the general implementation requirements are discussed in a later section.

5.1 REQUIRED METHOD STRUCTURE SUPPORT FEATURES

The TRIAD Model supports the representation of the structure of software engineering methods as follows:

- o The Unit Class provides "chunking" of method concepts and the tags of the Classes provide names for the software engineer to use,
- o Refinement links allow trees, hierarchies and graph based methods to be represented,
- o The Attribute provides storage for both long text strings and variables describing the method concepts,
- o Procedures to express method dependent knowledge based on the conceptual chunks of the method. Further, the procedures are invoked based on criteria such as access mode and type of entity requesting access (software engineer, tool, etc.) which are specified by the method definer.
- o A query language on Unit Classes, Components, tags, Attributes and Links allows fast access to stored text and fixed format data,

o Links to other units model secondary conceptual relationships,

5.1.1 CHUNKING OF CONCEPTS

The partitioning of a method into conceptual chunks is a natural way to subdivide a large number of entities. The Unit Class is used to represent a concept in a software engineering method. The Component Categories within a Unit Class serve to subdivide the concept into related pieces. Thus, the TRIAD Model initially provides a two level approach to the organization of concepts in a method. Further levels of detail can be introduced by the use of the Secondary Links.

Since many methods are representational, the TRIAD Model, facilitates the expression of these methods. Each Unit Class is a representational unit, say a box in SADT or a bubble in Dataflow Diagrams. Within the Class, the Component Categories describe the entities of the method. In the case of SADT, this includes the input, output, mechanism and control arrows and the descriptions of the boxes.

Even if the software engineering method is not representational, but procedural in nature, the TRIAD Model is still effective for expressing the method. Steps in the procedural method may be chunked together into one class, representing a task within the method. The key idea is to partition the method into workable and manageable entities.

Software engineering methods which are used to merely organize textual descriptions of software can be easily defined using the TRIAD Model. The Categories within a Unit Class are used to subdivide sections of the text. For example, one or more Unit Classes could be used to represent the documentation of a program. The Component Categories within the Unit Class would correspond to the major parts of the document. A method to store the requirements of a software project could be organized using the TRIAD Model by grouping similar requirements together. For example, one unit class may be for performance requirements, another for functional and so on. Of course, if there are no differences between the information describing performance and functional requirements than only one Unit Class is required.

Tags are attached to each Component Category in a Unit Class. The tags are used as names for the Component Categories, Unit Classes, Units, Entries, Attributes, Links and Procedures. The query language uses the tags as objects for searches of the information contained in a method.

Besides their use as reference strings, tags, when carefully chosen, can impart semantic knowledge to the user.

5.1.2 REFINEMENT LINKAGES

Each Category in a Unit Class may be a refinable Category. The refinable Category links are called Refinement Linkages, because they serve to refine a concept from a Component Category of one Unit Class to another Unit Class. The Refinement Links, which are the instances of the Refinement Linkages, are used to support the organization and chunking of concepts in the Units when the method is applied. The navigation through the Units for browsing or queries is done by using the Refinement Links as a default.

The Refinement Linkages are also essential in modeling the different types of graphical representations that are often found in methods. Hierarchies are simply modeled in the TRIAD Model by restricting each Unit Class to only one Refinement Link pointing to it. If a Unit has more than one Refinement Link pointing to it, then directed graphs are easily represented. Directed graphs are applicable to such methods as Dataflow Diagrams and program Call Structure charts. Although cycles of Units can be created, the processing of them may become complicated and the value of a method making extensive use of cycles might be suspect. If the intent of the method is to represent iterations through the software life cycle, then the version feature of the model implementation should be used to keep track of method iterations.

5.1.3 ATTRIBUTES

Attributes attached to the Component Category provide the means for storing as well as summarizing and describing. Several software engineering methods consist primarily of large blocks of text. For example, requirements analysis methods and documentation support methods dictate the content and procedure for accomplishing these respective tasks. The Attributes in the Component Categories of the Unit Classes for these types of methods often serve as repositories for the text. In this case, the Categories in the Unit Class are effectively used to provide further organization of the text. For instance blocks of text can be divided among the categories based on the method. A documentation method illustrates this point. Manuals are divided into sections and each section corresponding to a Unit Class may be further subdivided by the Component Categories. For example each command in the reference manual can be stored in a separate Entry of a Component Category. Such a structure imposed on the information by the method and supported by the TRIAD Model, greatly increases the retrieval of relevant information contained in the method.

The Attributes in addition to free form text, are used for fixed format values (integers, reals, booleans and words). In addition to being used to store values describing the Component Categories, the Attributes are used to implement many of the following special features of the software engineering method support, such as, Secondary Links, Procedure References and extended commands.

5.1.4 PROCEDURES

The TRIAD Procedure allows the TRIAD Model to represent local procedural knowledge about the contents of the Unit or an individual Entry. Coupled with the rule based invocation criteria, the TRIAD Procedure supports methods by providing a means to encapsulate local knowledge such as the design rules in the Jackson Method. A Procedure could be written to diagnose a structure clash and perhaps suggest alternative designs to avoid the clash. The Procedure is provided to the method definer as a means for customizing a method specification. It is also the vehicle for storing predefined queries, tool interfaces and the definition of simple automatic processing steps.

5.1.5 QUERY LANGUAGE

The greatest difficulty in processing information stored and produced by the application of methods is the strong reliance by many methods on natural language text. Text strings are difficult and time consuming to process and usually can only be searched by examining each character individually. The query language is important for supporting methods because the ability to formulate queries and quickly retrieve information is the expected benefit of encoding and keying information into a computer. The query language is the major vehicle for utilizing the stored information contained in the TRIAD Model Units. Some example queries using the Call Structure example include:

- o List all modules in the system,
- o Find all modules rated as difficult to implement,
- o Show all modules not yet completed,
- o Compute the number of man-hours expended over the estimate and
- o Display a graph of module completion dates (actual vs. estimated).

Although the quality of a query language is largely implementation dependent, the TRIAD Model has been developed with the objective of supporting a robust query language easily. The TRIAD Model supports this diverse sampling of queries by allowing:

- o The creation of tags to name components (to use as objects of queries),
- o Attributes to store method definer specified values,
- o Procedure references to process attribute (all under method definer control),
- o Refinement links to navigate through the Units of the software engineering method for logical and faster searching and
- o Secondary links to other units to improve navigation performance and to search the information contained in the method based on secondary relationships.

The actual syntax of the query language is not dictated by the TRIAD Model and the design is left up to the implementor. However, the syntax of the query language should be easy to use especially for casual and novice users. In addition, it should still be powerful enough to satisfy the expert user.

A list of available commands can be easily extracted by the query language using the above example. Or the description of a particular command can be extracted by the query package by searching all entries of the command component category for the specified command name and displaying the accompanying command description when the name is located.

5.1.6 SECONDARY LINKS

In addition to the Refinement Linkages, which are used as the primary organization of Unit Classes, additional Secondary Links can be defined and used to describe relationships other than refinement. One use of Secondary links is to tie all Units of the same Unit Class together. Each Unit can then be processed by merely following the Secondarv Links. A more complicated use of Secondary Links would be to create alternate paths through the refinement graph. Another example of Secondary Link usage is to bind requirements documents to actual software code which will be developed later in the project. By this use of Secondary Links, it is possible to associate requirements created in the initial project phase (and created by a different method) to software designs (and eventually code) created later in the software life cycle.

5.2 REQUIRED IMPLEMENTATION FEATURES

The requirements for an implementation of the TRIAD model given in Chapter II are:

- o Easy to use interface,
- o Efficient and fast storage and retrieval of Entries and Units,

- o Graphic views of Units and their Refinement Link structures,
- o Robust and easy to use text editor and
- o Flexible tool interface.

5.2.1 USER INTERFACE

Much of the user interface is dependent on the implementation vehicles and the implementor; however, the TRIAD model encourages the organization of the user interface about the Component Categories and Entries. It is intended that the Component Category will usually be a compact entity in the method. Further, the corresponding Entry, when filled out, should fit on a single display screen. The Attributes are associated with the Component Categories and their values with the corresponding Entries, therefore, the commands, help and tutorial services should be similarly organized about the Component Categories and Entries. Such a design will help the method definer to create extended commands that are associated with the Component Category that is the source (or target) of their operation. (Extended commands are also defined in the same manner as procedures. The difference between the two is that extended commands must be explicitly invoked, usually by the user. Procedures as already described, are invoked indirectly based on the user's actions.

5.2.2 STORAGE AND RETRIEVAL MECHANISM

The TRIAD model consists of only a few basic elements which must be stored. This feature facilitates the use of either a database management system or physical storage scheme. The Component Categories and Attributes are the two entities that must be stored for the Method Definition Component. The Unit Classes and Refinement Linkages are constructed by relations or pointers. The Secondary Links can be implemented as Attributes. Similarly, the Entries and Attribute Values are the two basic elements of the Method Use. The Refinement Links, Component membership and Unit membership are constructed from relations or pointers.

A graphic interface package is supported by the TRIAD model by simply transforming the Units into icons and the Refinement and Secondary Links into arcs. The placement of the icons on the screen in a left to right, top to bottom sequence is dictated by the sequence of the Entries which refine to Units within each Unit beginning with the Initial Unit.

5.2.3 TEXT EDITOR

Text editor support in the TRIAD model is accomplished by clearly delineating the text from the method structure. Text is stored in Attributes associated with Entries. This separation permits the text editor to be invoked upon a text string contained in an Attribute much as any external tool. After the user is done editing, the text is replaced and control is returned to the implementation for the next user action.

5.2.4 TOOL INTERFACE

To effectively use existing tools, the TRIAD model allows tools to be invoked without direct Method Use requests. This is accomplished by treating the tools as Procedures and using the rule based invocation feature of the Procedures to call the tools.

Further, the naming of the Attributes and the separation of the Attribute values from the method structure, allows the user to extract (or insert) information in the Method Use by using the Attribute name and calling an implementation provided primitive routine to do the extraction (or insertion).

Batch tools are easiest to int·grate because the data can be extracted, the tool invoked (control relinquished), and the results replaced (if necessary). Interactive tools follow the same sequence but many times over, The ease of integrating interactive tools depends largely on the facilities provided by the operating system on which the model is implemented.

The TRIAD Model supports tool interfacing by providing data access routines and a comprehensive facility for invoking tools.

5.3 MULTIPLE SOFTWARE ENGINEERING METHODS SUPPORT

There are two ways to provide support for multiple software engineering methods. The first technique is to provide translators from one method to another. In addition to the effort involved in writing these translators, the difference in representation between the same concept in different software engineering methods poses a difficult task for the translator. For example, a data oriented software engineering method, such as Dataflow Diagrams does not directly map to a Call Structure Chart. Different data elements may have separate processing bubbles, but the system structure can aggregate all of the processing in one module.

The translators must be bi-directional, since the cyclic nature of software development may require that if an error is discovered in the coding phase and fixed there, then the correction should be reflected in the program design and system design. Theoretically this should not happen, because an error detected in the coding phase should cause a change in the program design first and then the coding change. The reality of the situation is that designs are updated after the fact (if at all). This is primarily true if the program coder was not the designer. From the coder's point of view it is faster to make the change first (especially if it is a small change) and then update the design later.

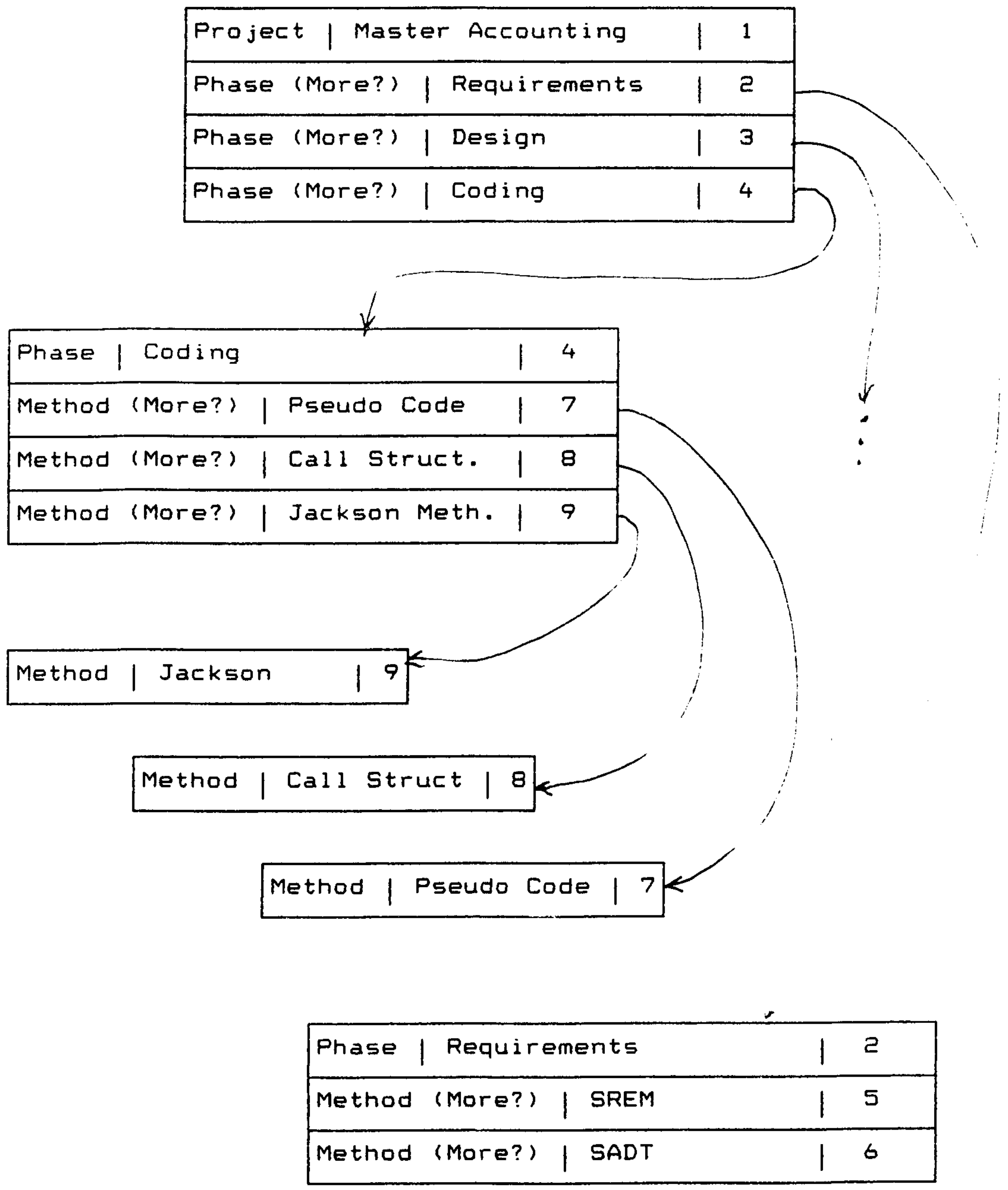
If more than one software engineering method is used for a particular phase, for example, Jackson Method and pseudocode for coding, then these translators would be run constantly to keep the software representation current in both methods.

The second technique for supporting multiple software engineering methods is to use a common representational scheme. The most direct approach of this technique is to use a database management system to store all of the project data, including the method representations. Some methods, notably PSL/PSA, claim to have accomplished this, and can support all methods [TEIC77]. In fact several popular methods have been implemented using PSL/PSA. Extensions to

PSL/PSA provide dictionary features and support routines. A meta-language processor allows a language based method to be specified. However, PSL/PSA is still a language based Database Management System approach to method specification. It is unclear how effective PSL/PSA is as a specifier of software engineering methods when it is itself a software engineering method. [CHIK85] The TRIAD Model is a much more general mechanism for method representation then PSL/PSA.

This database approach depends on the selection of an appropriate database system that uses a data model which is capable of representing software engineering methods easily and completely. Chapter IV has already discussed the problems with using database management systems to support software engineering methods.

The TRIAD Model supports the second technique of multiple method support by using a model specifically designed for methods. Each method is defined as separate Units. Secondary Links between different methods and Procedures can be used to translate Entries and Attributes between methods. Of course the specification of the appropriate link types would still need to be done by a human, the method specifier. However, the environment generated from the TRIAD Model specification of the method would do the translation dynamically. This feature makes it very easy for a software engineer to switch methods and view the same software in a different way. Figure 15 shows a possible arrangement of several software engineering methods. The methods are organized around a software life cycle model. For each Unit Class representing a phase are several subordinate Unit Classes each representing the Initial Unit Class of a different method.



Project | Master Accounting | 1
Phase (More?) | Requirements | 2
Phase (More?) | Design | 3
Phase (More?) | Coding | 4
Phase | Coding | 4
Method (More?) | Pseudo Code | 7
Method (More?) | Call Struct. | 8
Method (More?) | Jackson Meth. | 9
Method | Jackson | 9
Method | Call Struct | 8
Method | Pseudo Code | 7
Phase | Requirements | 2
Method (More?) | SREM | 5
Method (More?) | SADT | 6

Figure 15. Multiple Software Engineering Methods

CHAPTER VI

IMPLEMENTATION OF THE TRIAD MODEL

Although the focus of this dissertation is on the model for representing software engineering methods, the model was implemented to verify its design and to demonstrate the use of the model. This chapter describes aspects of that implementation. An understanding of the implementation is not necessary to understand the model, therefore, this chapter may be skipped by the reader who is not interested in the implementation.

The implementation of the TRIAD model represents a large piece of software containing several thousand lines of source code. Rather than describing the actual implementation in detail, this chapter presents interesting problems encountered during the implementation. The solutions and reasons for the solution are also given. The complete implementation is described in the documentation method of the TRIAD multiple software engineering method. The TRIAD software engineering method is described in the next chapter. The TRIAD model operators defined in Chapter III provide a detailed description of the necessary functions that must be provided to adequately fully the

TRIAD model.

The Grammar Form Model was used as the basis for an implementation of a method specification and environment generator called TRIAD. This implementation was done on a DEC VAX using the C programming language running under the UNIX operating system. (DEC and VAX are registered trademarks of Digital Equipment Corporation. UNIX is a registered trademark of Bell Laboratories) This implementation of TRIAD had a strong grammar orientation. The method specifier had to enumerate all of the symbols (tags) and the production rules manipulating the symbols to create forms for a method. Under a contract from IBM, the TRIAD concepts were implemented on an IBM 4341 computer running VM/CMS. To quickly implement TRIAD, an interpretive programming language REXX [IBMR] and the system editor [IBMX] were chosen as implementation vehicles. Learning from the UNIX implementation experience, the VM implementation abandoned the Grammar Form Model, especially at the user interface level. The method specifier directly manipulates the Component Categories and Unit Classes rather than productions and symbols to create entities representing method concepts.

6.1 IMPLEMENTATION VEHICLES

The interpretive language, REXX, was chosen for the IBM implementation because it was designed to work closely with the editor, XEDIT. In fact, it was possible to invoke XEDIT from REXX and to issue editing commands within a REXX procedure. Since a major part of a software engineering environment is a text editor, this design decision eliminated the writing of an editor. Of course the resulting implementation was slower than if TRIAD had been implemented using a compiled language such as PL/I or PASCAL, however, the concepts embodied within the TRIAD model were adequately demonstrated.

Since XEDIT was accustomed to working on entire files, each Unit and Unit Class is stored as a separate file. Chapter VIII discusses alternative methods of storing the Unit Classes and Units.

XEDIT has several features which greatly facilitated the implementation of TRIAD. Each line in a file being edited by XEDIT can be assigned an integer representing its display level. By setting a global display range, only those lines whose display level falls within the range will be visible. This feature allowed the mixing of TRIAD control lines and method specific text with the entries. The TRIAD control lines were assigned a different display value then the method lines. For user displays, the display range was set to just the method lines. If a TRIAD REXX routine was manipulating the file, then all lines would be made visible (only to the REXX routine, the screen display is maintained until the REXX routine exits). Although this technique is not generally applicable, since it depends on an esoteric feature of the editor, it did simplify the storing of the structure and the actual data by allowing the two types of data to be stored together in the same file.

The second valuable XEDIT feature was the label facility. Eight character labels can be assigned to any line in a file being edited by XEDIT. Thereafter, these lines can be referenced directly by using the labels. This feature was used extensively to jump directly to a specific entry on the screen display, thereby eliminating time consuming free string searches.

6.2 SYSTEM ORGANIZATION

The implementation is loosely divided into three major groups of routines: Tuner or Method Definition Component, Editing or Method Use Component), and System Integration Library (common sub-routines). Since TRIAD operates under XEDIT, each command is implemented in REXX as a separate routine, stored in a separate file. The best way to view the function of the TRIAD components is to look at the commands implemented.

The Tuner contains commands to create Unit Classes and Component Categories within the Unit Classes. Commands also exist to modify existing method specifications. The Tuner commands have been already described in Chapter III as the TRIAD model operators.

The editor provides similar commands for the creation of Units from the Unit Classes specified in the software engineering method. The focus of this dissertation is on the model for representing software engineering methods. The editor merely creates instances of the Unit Classes defined using the Tuner, therefore from a conceptual point of view, the elements of the model are all covered in the Method Definition Component. Thus, a detailed discussion of the editor is not within the scope of this dissertation.

6.3 TUNER SUPPORT FEATURES

To help the method designer create a method specification, TRIAD maintains three lists. The first list is the names of all the Unit Classes defined. The second list is all of the Unit Classes referenced, but not yet defined. These lists are used by TRIAD to insure the uniqueness of the Unit Class names. The lists are also helpful to the method designer, who can specify a command which displays the lists on the terminal screen for reference. Thus, if the method designer is defining the method top down, a display of the undefined list will show the names of the Unit Classes that must still be specified.

When the method specified using TRIAD is applied using the TRIAD Method Use Component, the list of Unit Classes shows all the Unit Classes defined. A third list is created when a method is applied which contains the names of each Unit, its Class and serial number. This list is used by the environment to efficiently process the Units. As with the other two lists, this list is also a valuable reference for the software engineer applying the method. It summarizes the method use by displaying in one place all of the Units, which is particularly useful for a software engineer who is just browsing. Figure 23 in the next chapter is one example of this list.

6.4 HARDWARE FEATURES

TRIAD was designed to use an IBM 3279 terminal which is a synchronous, color terminal. It has a standard typewriter style keyboard with additional keys for cursor movement and screen display control. Twelve function keys are also on the keyboard which can either be bound to command strings or detected by REXX programs as special function keys.

Since the terminal is synchronous, an entire screen of data is transmitted each time the enter key is pressed. Cursor movement is under local terminal control and cannot be detected by a program executing on the host computer. This characteristic of the terminal makes protection of screen fields and the tracking of cursor movements difficult if not impossible. However, by using the protection feature of the 3279 terminal, which is under program control, the user's editing actions can be limited to only program designated areas of the screen.

XEDIT divides the screen into several blocks of lines consisting of the following:

o Status line - information about the file currently being edited,

o Message lines - space to display error or other messages from the editor or REXX programs. This space can overlay the file area,

o File area - block of lines where the edited file is displayed and changed,

o Current line - a line within the file area which is the default target for all line oriented editing commands,

o Reserved area - block of lines within the file area reserved by XEDIT commands. The user cannot change this area and

o Command line - Line to enter editor commands.

TRIAD uses these blocks as follows to create a useful display for software engineering methods support.

- o Status line - changed to show the user TRIAD specific information such as the number of Secondary Links, Attribute and queries attached to the Entry under the current line,
- o Message lines - placed as an overlay at the top of the file area. The superimposed message can be cleared by pressing the enter key and the original screen display will be uncovered,
- o File area - used to display the Unit Class or Unit. It is kept as large as possible to minimize user necessitated screen scrolling,
- o Current line - retained in the center of the screen,
- o Reserved area - Three lines are reserved at the bottom of the file area. The first two lines display the commands bound to the function keys and the third line shows the names of alternative menus which contain different key bindings and
- o Command line - Retained at the very bottom of the screen.

Figure 16 shows the screen layout.

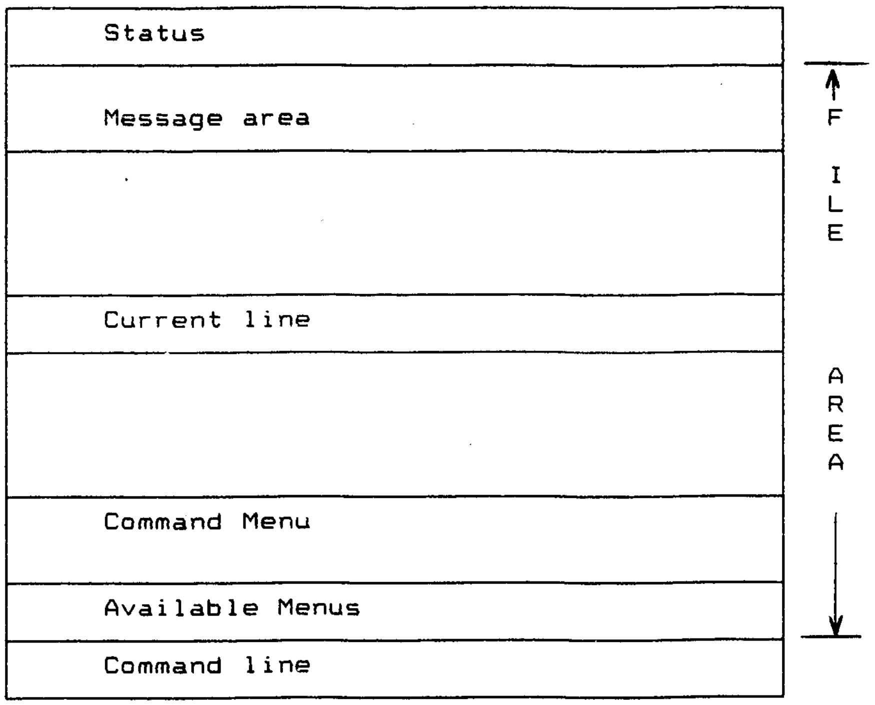


Figure 16. TRIAD Screen Layout

6.5 VISIBILITY

A difficult problem with any computer system is organizing the display such that the right information is available for inspection by the user. Since the display is limited to the finite size of the computer terminal, it is not always possible to fit all of the information on the screen at one time. Further, it is difficult to filter out information without destroying the user's perception of the structure of the information being displayed. This problem is best illustrated by considering overlapping information within software engineering methods. For example, a program coding method may record information about the program and its development progress such as start date, estimated completion date, size, estimated size, etc. This information is of primary interest to management and should reside in a management method. However, the software engineer generates the information and has a right to have access to it. The approach taken by TRIAD to solve this problem is to replicate the information in both methods (program coding and management) and use Procedures to propagate a value whenever it changes. While this solution solves the access problem to overlapping information, the display problem still remains.

When the software engineer is involved in coding, the presence of the management information is unnecessary. To temporarily hide information, TRIAD attaches an Attribute called "VISIBLE" to each organizer. This Attribute contains a single value which must match the user set visibility mode. A visibility mode of ALL causes all organizers to be displayed regardless of their VISIBLE attribute value. This feature allows the software engineer to restrict the display to only coding related organizers while doing coding, thereby simplifying the display.

6.6 GRAPHICS SUPPORT

The TRIAD VM implementation uses graphics to present to the software engineer a pictorial view of the software engineering method and the resulting software application. [HART87] (The TRIAD graphics interface was implemented by Ronald Hartung) The graphics interface is implemented using GDDM [IBMG] and operates on an IBM PC/GX synchronous terminal.

The simplest use of graphics in TRIAD is to draw graphical images on the screen and allow the user to store them in an Entry for subsequent display. This feature allows graphical images to be integrated with text, which is good for documentation methods.

The primary use of graphics in TRIAD is to provide the software engineer with pictorial views of the method (Unit Classes) and Units. Each Unit Class has an Attribute which defines an icon to represent it. The icons can be designed by the method designer using the GDDM based iconic editor. By invoking a command to draw a graph of the method, the TRIAD graphics interface uses the icon definitions and refinement links to produce a graph of the method. In addition commands are provided to manipulate the display by zooming and panning. Further, a Unit or Unit Class can be selected for display in the normal text mode, thereby allowing the software engineer to view the entire method's Units as a graph and edit it is a textual unit. The TRIAD query language (TMQL) can also be used to select a region of the method which is then displayed by the graphics interface as a graph.

6.7 STORAGE AND RETRIEVAL OF TRIAD MODEL ENTITIES

Currently TRIAD stores each Unit Class and Unit in a separate file. Since VM does provides data protection only at the file level, and TRIAD should protect the Unit at the Entry level, an alternative means of protection is needed. Using the VM file system TRIAD provides just read-only access to methods and instances of methods stored on a different disk from the users. However, any modifications to a Unit Class or Unit are made on a copy of the Unit Class or Unit and stored on the users disk. Since database management systems have solved the multi-user access protection problems, a suitable database management system was sought. The IBM relational database product, SQL was used to implement a storage and retrieval facility [DAVE86]. Relations were created to hold the Entries and Attributes.

Since response time was already long, the use of the SQL database management system exacerbated the condition. Chapter VIII discusses possible solutions to this problem that need further investigation.

6.8 TRIAD MODEL QUERY LANGUAGE

An important feature of an software engineering environment is the ability to query the stored information. The user of a method wants to query on the structure of the information contained in the model as well as its content. Queries can be constructed to search only Entries of a specified Category (tag) in Units of a specified Class. The query language, TMQL, is modeled on SEQUEL, where a query can be just on the structure of the Units (maps directly to the SQL relations) then it is passed directly to SQL for processing. In other more complex queries, a TRIAD query processor parses the query into two parts--structure and content. The structure part is generated as a SQL query and the results of the query are searched for the content part of the query by TRIAD.

6.9 TOOL INTERFACE

Tools are invoked either explicitly by the user the same way an extended command is, or automatically as a TRIAD Procedure is. The means and criteria are controlled by the method definer. In either case, the tool interface for any tool is created using the TRIAD Procedure facilities. All of the operators are available to extract data from the Attributes and then invoke the external tool.

Tools are generally one of two types--Batch or Interactive. Batch tools are the easiest to create interfaces for. The data is extracted from the Attributes, placed in a file and the tool is called. Upon completion, any output is returned to the appropriate Attributes. Of course, for large volumes of data from many Attributes, such an interface can be quite large and cumbersome, but not complicated.

An interactive tool that requires data from the Attributes interleaved with user responses, is much more difficult to interface. If the host operating system which the TRIAD model is implemented under, supports a filter between the user and the tool, then this type of tool can be interfaced. For those operating systems that do not support a filter, the tool must be abandoned or an extensive amount code be written to simulate the tool's interactions. The user responses can then be placed in a file and the tool invoked as a batch tool. Of course, if partial computations are made based on the user input then this method probably would not work either.

It is important to note that the problems with interactive tool interfaces are not specific to the TRIAD model, but occur when interfacing any interactive tool to another system.

To support software engineering methods, an interface from the TRIAD generated environment to existing tools is essential. Such an interface facilitates the use of existing tools without re-coding them to work within TRIAD. The tool interface was demonstrated in the TRIAD implementation with several tools.

The Document Composition Facility (SCRIPT) [IBMD] was simply integrated by creating an Attribute which indicates the text stored in the Entry is SCRIPT input. A Procedure was then written to extract text from the Entries (an SIL provided function) and invoke SCRIPT. The output from SCRIPT was sent directly to the printer, although it could be returned to an Entry within the Unit specified for holding formatted output. The addition of special attributes to contain SCRIPT commands, which for instance, when used by a Procedure, extracted the Entry name (another SIL provided function) and made it a heading using the SCRIPT heading command. This approach can be expanded by using generic formatting commands in the Attributes. The

Procedure extracting the text will use a translation table to perform the translation from generic to specific formatter commands. This approach creates independence of formatter, allowing not just SCRIPT but other formatters to be used.

Another use of the tool interface was to extract source code entries and send it to a compiler for processing. This interface for PL/I also extracted the error messages from the source listing, which in PL/I are placed together at the end of the source listing. The messages were positioned following the offending statements and placed into an Entry created for the purpose or holding the source listing. The programmer is thus provided a compact view of the program and any compilation errors.

6.10 TRIAD PROCEDURES

Several different uses for Procedures were discovered during the application of the multiple software engineering embodied in the TRIAD method. The first use of a Procedure was the propagation of the tag from the Unit to the Entry refining to the Unit. For example, the software engineer may create a "MODULE" Unit Class for each module in a Call Structure Chart. In the header of the "MODULE" Unit Class is a space for the module name. Upon exit from the creation of an instance of the "MODULE" Unit Class, a Procedure is invoked to copy the name of the module to the Entry refining to the module Unit (See Figure 17).

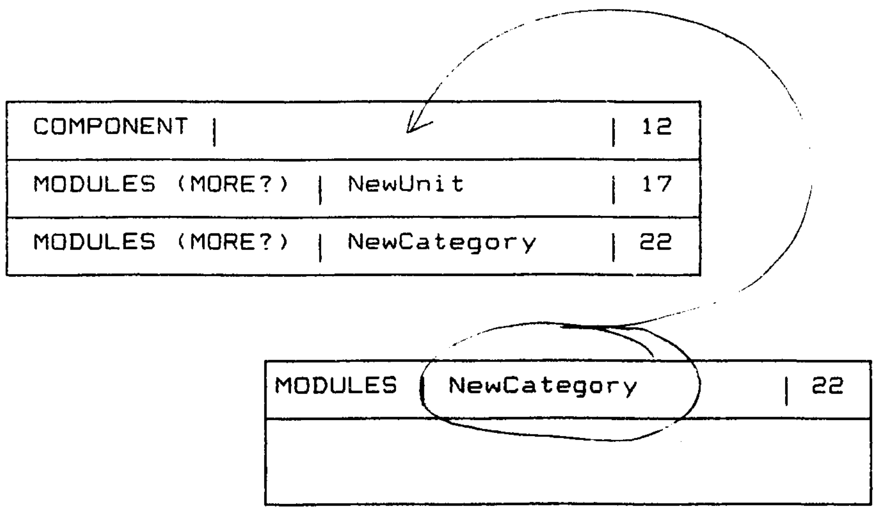

COMPONENT | | 12
MODULES (MORE?) | NewUnit | 17
MODULES (MORE?) | NewCategory | 22
MODULES | NewCategory | 22

Figure 17. Example of Information Propagation

The next use for a Procedure was necessitated by the TRIAD implementation vehicles. Since Rexx is interpreted and Xedit invokes a Rexx routine by searching for a file of the same name and of type XEDIT, the Rexx source could not be stored in a Unit and still be executable. The solution was to create a Procedure called PULLCODE which is invoked whenever the user positions the Cursor on "source code" Entry. PULLCODE uses an Attribute associated with the Entry to obtain the file name and file type. With this information it inserts the file into the Entry's text area. When the Entry is exited, a Procedure is invoked which provides the user with the option of saving or discarding the inserted code.

Procedures were also used to create syntax template editors for Rexx and PL/I. The editors are invoked by entering an Entry with the source code Attribute, which gives the name of the source code compiler. Templates are bound the terminal function keys and a menu is displayed showing the bindings and the statement types generated by pressing the different function keys. The user merely positions the terminal cursor at the appropriate place to insert a language structure and presses the appropriate function key to generate the desired structure.

Procedures are also used to automatically update the TRIAD help system based on the user modifying the system documentation. The insertion of a new command in the list of commands (LISTOFCO) Unit causes a Procedure to be invoked which inserts the new command name and command description obtained from the Entry into the TRIAD help system.

The TRIAD method was applied to the development of TRIAD. In particular, the MAJORCOM (major component) and MODULES Unit Classes were used to partition the many TRIAD routines into appropriate categories, and thus represent a system Call Structure Chart.

The use of TRIAD provided much insight into the interface design and the usage problems created by adding semantics to the TRIAD model. Making the TRIAD user aware of existing Attributes, Secondary Links and commands (queries) was one problem discovered through the use of TRIAD in TRIAD.

6.11 USER INTERFACE

A key issue in the construction of any software today is a good user interface. Often referred to as "user friendly", the goal with TRIAD was to produce an interface so that the user would never be in a quandary on how to accomplish the next task.

The section on hardware features described the screen layout, which was crafted so that the user would see the list of available commands bound to the function keys from which to choose the next command. Since TRIAD is a user active type of system (the user must enter a command rather than selecting from a menu or answering dialogue questions), the menus are vital to keeping the user aware of available options. The menu and function key binding concept is carried a step farther, by changing the bindings and menu based on the user's previous action. For instance, if the user's previous command was to create a new Component

Category, then the bindings and menu would be set to those commands to edit (add/delete Attributes, Secondary Links and Procedure references) a Component Category. The ultimate purpose of this feature is to only present the user with those commands which are valid (based on previous actions and current display) and to anticipate the next likely command.

TRIAD commands are designed to be single action and not have any parameters. If parameters are required, then the REXX procedures implementing the command will solicit the required parameters from the user by way of a question and answer dialogue.

Help with commands is provided to the user in two ways. If the user knows the name of the command then entering HELP followed by the command name will produce a brief description of the command in the message area of the screen.

The other help system is modeled after the CMS help system and presents the user with a table of all available commands from which the user selects one by placing the cursor over it and striking enter or PF key 1. The command description is then displayed on the screen.

CHAPTER VII

DEMONSTRATION OF TRIAD MODEL

Several software engineering methods were used as the basis for establishing the requirements for a model to represent methods. In this chapter, two methods will be defined using the TRIAD model to verify the design of the model and to demonstrate the effectiveness, completeness and support features of the model. The Jackson Method will be defined first. The example from Chapter II will be used to show the application of the TRIAD defined Jackson Method to a software design problem. The second example is a multiple software engineering method developed to support the development of the TRIAD model implementation. Each of the multiple methods is briefly described and two of the methods are shown in detail.

7.1 JACKSON METHOD

Jackson Method, as described in Chapter II uses a few symbols to create a view of software (data and control structures) which enables the software engineer to design modules. The objects of the method are boxes and lines which are arranged hierarchically to reflect a top down, left to right order. The boxes are used by the designer to partition the module or data structures into groups of processing actions or substructures, each represented by a box. The lines between the boxes are used to arrange the boxes into a hierarchy. Three types of boxes are possible in the Jackson Method, each representing a different processing or data structure construct. The boxes are differentiated by the presence or absence of a symbol in the upper right hand corner of the box. A box with no symbol represents a sequence of processing actions or data elements and the actions are done according to their position in the hierarchy or the data elements are ordered according to their position in the hierarchy. A box with an asterisk (*) in the upper right hand corner corresponds to an iteration such as a DO or REPEAT statement in a programming language. For a data structure, the iteration box represent a repeatable data structure or an array of data elements. IF-THEN-ELSE or CASE statements are represented by the selection box which is signified by a small circle in the upper right hand corner of the box. The selection box for a data structure represents several data structures redefined on the same space. For instance, the REDEFINES statement in COBOL or the VARYING CASE RECORD in PASCAL are examples of the selection for data structures.

To define the Jackson Method using the TRIAD model, the method objects are matched to the model elements. Each box type is defined as a separate Unit Class. Since the boxes are similar, the sequence box will be discussed in detail. The initial Component Category contained within the sequence Unit Class will have space for the name of the box. An Attribute of type text will be associated with the Category to hold the description of the processing actions the box represents. Another Attribute specifies the name of the shape of the symbol, in this case a box, that the user interface will display. A second Component Category is defined to contain the line between this box and the children boxes, which will be represented as Units of the correct Unit Class. The Refinement Link from this Entry to another Unit can be to any Unit of the three classes. This Component Category is repeatable so that any Unit can have more than one Entries in the Component which will create Units subordinate to the Unit containing the Entries.

Figure 18 shows the user view of the Unit Class for the sequence box as created using TRIAD. The Attributes are not directly visible to the user and are present to show their relationship to the Component Category. The visible text is shown in bold type.

```
Attributes: (icon;box)
Attributes: (description;text)
Sequence |                            | Unit Number
---------------------------------------------------
Attributes: (repeatable;)
Attributes: (refines to;box, obox or *box)
Subordinate Proc. |              | Refinement Link
```

Figure 18. Sequence Unit Class for the Jackson Method

Returning to the name and address file example of Chapter II, the application of the Jackson Method defined using TRIAD would produce a tree of Units as shown in Figure 19. Each Unit is shown in this figure as the user would view it, i.e. the Attributes are not shown. This figure shows the TRIAD model definition of the Jackson Method. The graphic package will be able to display this example using the familiar Jackson boxes as was originally shown in Figure 7. Such a graphic interface would be essential to successfully use the TRIAD model for the Jackson Method, since the arrangement of the boxes is critical for the user to understand the design.

Sequence	Process Transaction	Unit Number 1
Subordinate Proc.	Valid	Refines to 2
Subordinate Proc.	Invalid	Refines to 3

Selection	Valid	Unit Number 2
Subordinate Proc.	Update File	Refines to 4
Subordinate Proc.	Print Rpts.	Refines to 5

Selection	Invalid	Unit Number 3
Subordinate Proc.		Refines to

Selection	Update File	Unit Number 4
Subordinate Proc.		Refines to

Selection	Print Reports	Unit Number 5
Subordinate Proc.		Refines to

Figure 19. TRIAD Application of Jackson Method

This example shows that the TRIAD model does generally represent the Jackson Method using the three Unit Classes each consisting of two Component Categories and appropriate Attributes. Further, Attributes describe the Unit Class as an icon so that the user can see and manipulate the method graphically. By applying the TRIAD defined method to a software design problem, a representation of the software in the Jackson Method is quickly realized.

This example merely demonstrates that the TRIAD model is capable of representing at least one method. The next section expands the use of the TRIAD model to several methods and demonstrates how features of the model such as Secondary Links and Attributes can be combined with the implementation to assist the user in the development of the software.

7.2 THE TRIAD METHOD

A TRIAD generated multiple software engineering environment for developing software was created and used to document the TRIAD implementation. This method, called the TRIAD Method, was loosely based on standards used to maintain existing software (a relevant example to the research sponsor). The method focused primarily on software coding and testing, but also contained Unit Classes dedicated to requirements, documentation and management.

The following software engineering methods are contained in the TRIAD Method:

- o Life Cycle,
- o Documentation,
- o Management,
- o Requirements,
- o Program Structure,
- o Pseudo Code and
- o Coding

Table 14 shows each Unit Class, its corresponding method and a brief description for the TRIAD multiple method.

Table 14. TRIAD method Unit Classes

Unit Class	Method Supported	Description
PROJECT	Software Life Cycle	Start Unit Class
PHASEO	Software Life Cycle	Project Objectives
PHASEI	Software Life Cycle	Overall Architecture
PHASEII	Software Life Cycle	Programming Logic
MEMBER	Management	Project Participants
SCHEDULE	Management	Schedule
REVIEW	Management	Review
HISTORY	Documentation	History of Project
USERSMAN	Documentation	Users Manual
INTRODUC	Documentation	Introduction to Manual
TERM	Documentation	Terms used in Project
USAGEEXA	Documentation	Usage Example
LISTOFCO	Documentation	List of Commands
FUNCTION	Requirements	Functional Overview
FUNCCHAR	Requirements	Functional Characteristics
CONFIGUR	Requirements	Configuration
RATIONAL	Requirements	Rationale for Design
HUMANFAC	Requirements	Human Factors
MAJORCOM	Program Structure	Major Component
MODULES	Program Structure	Modules
LIBRARY	Program Structure	Library of Modules
PROSEPRO	Pseudo Code	Prose Prolog
MAKE	Coding	How to Compile and Link
DATASTRU	Coding	Data Structure

The TRIAD method is primarily a life cycle model, which was used to organize the software development process. The Initial Unit Class is the Project Unit Class which owns the classes for the other methods. The first method is the software life cycle and is represented in the Unit Classes PROJECT, PHASE0, PHASEI and PHASEII. Phase 0 is the project objectives and reflects the initial planning for the implementation. Phase I describes the overall architecture of the TRIAD environment and represents the system requirements, design and reasons for the design. Finally, Phase II is the programming logic of the implementation.

A management method is represented in several Unit Classes by capturing data relevant to the process of creating software. Rather then being an isolated set of Classes, these Unit Classes are referenced throughout the other method Unit Classes by refinement links. The Unit Classes MEMBER, lists all of the project participants and their addresses and phone numbers. The Unit Class SCHEDULE is used to track the time and effort expended on the software development. Finally, the REVIEW Unit Class is used to summarize the project meetings and record the progress on the software development.

The documentation method consists of four Unit Classes which describe the composition of the users manual. An additional Unit Class is used to record the history of the project. It has content that is both of value as documentation and also for the management of the project. The documentation method Unit Classes are HISTORY, USERSMAN, INTRODUC, TERM, USAGEEXA and LISTOFCO.

The SCHEDULE and REVIEW Unit Classes are part of a management method, because they record data on the progress of the software development. This information can be used by the project managers to make decisions concerning the progress of the development and take actions to solve any problems identified by the information contained in the units.

The requirements method is composed of the Unit Classes FUNCTION, FUNCCHAR, RATIONAL, HUMANFAC and CONFIGUR. Each of these classes focuses on particular requirements of the software, namely, overall functions, functional characteristics, rationale behind the design, human factors and system configuration.

The software production is supported by three methods--program structure, pseudocode and coding support. The program structure method organizes the software major components (MAJORCOM), libraries and modules, with a Unit Class corresponding to each organizational type. A major component is composed of modules as is a library, however, the library is used to store common routines, while the major component represents different processing sections of the project. The Refinement Linkage from a major component into a modules Unit Class represent the calling of a module. The refinement linkage from a LIBRARY Unit Class into MODULES a Unit Class represent the inclusion of the modules into a library, which is a group of similar functions under a general category, such as Tuner commands.

Coding support is provided through the Unit Classes MAKE and DATASTRU which contains information to help the software engineer code and debug the source code. The MAKE Unit Class details the steps necessary to create executable code from the various modules and libraries. The DATASTRU, data structure, Unit Class is owned by the MAJORCOM Unit Class and is used to describe all of the significant data structures used by the modules within the major component.

Finally, the PROSEPRO Unit Class supports a pseudo code language method for describing a module's function. This Unit Class is used much like a documentation method, except it contains documentation on the construction of the module. This Unit Class is owned by the MODULES Unit Class.

Figure 20 shows the refinement link structure for the multiple software engineering method used in the development of TRIAD. The refinement links are represented by the arrows that constitute the ownership of one Unit Class by another. Although this method is hierarchical, there is no restriction imposed by the TRIAD Model or TRIAD Environment that it must be.

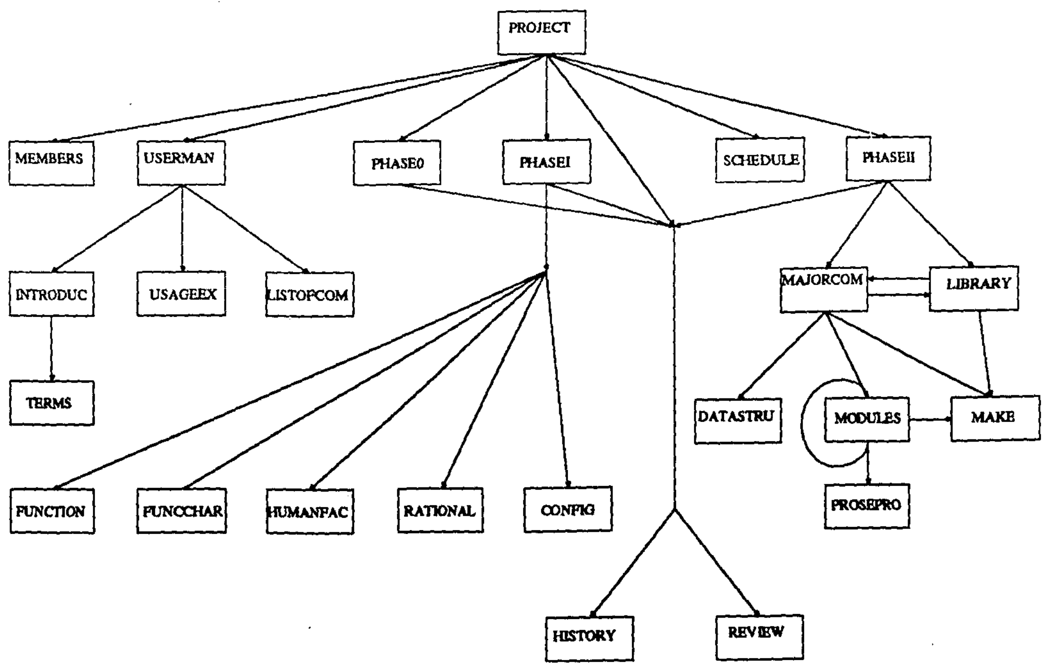


Figure 20. TRIAD Multiple Method Unit Class Refinements

From Figure 20 the relationship between the various methods is apparent. Most of the methods are in small groups of Unit Classes clustered together. For example, at the top of the figure, the software life cycle method organizes the rest of the methods. At the left, is the documentation method with a small tree of Unit Classes representing the users manual. Next to the documentation method is the requirements method with several Unit Classes serially owned by the PHASEI Unit Class. Similarly, the program structure method is owned by the Programming logic Unit Class, PHASEII. Only the management method Unit Classes, HISTORY and REVIEW are attached to most of the high level Unit Classes and not organized into its own hierarchy. Further evolution of the management method would indeed contain some independent Unit Classes that would contain summarized data, such as a complete project schedule. But the current management classes are used to contain data at the point of creation.

7.2.1 DOCUMENTATION METHOD

The documentation and program structure methods will be used to illustrate the use of the TRIAD Method. Figure 21 shows the user view of the Unit Class USERMAN. Each

Component Category is separated by a solid line. The first Component Category is the one for the Unit Class and contains the method and Unit Class name with space left in the center for a descriptive string to be entered whenever a new Unit is created. On the right is the Unit's serial number. The next four Component Categories are for the storage of text strings describing the topic suggested by the Component Category name. The following 3 Component Categories can each be replicated which is indicated by the string "(MORE?)" appearing at the far right. In addition to being replicated individually, i.e., a Section may be composed of more than one topic, the indentation of the category names indicates that the entire structure may be replicated by requesting the replication of a higher level category. For instance, a document can consist of several sections each with at least one topic. More than one document can be stored in this unit, each with at least one section, consisting of at least one topic per section.

```
-------------------------------------------------------------
| TRIADMD-USERMAN |                             | UNIT       |
-------------------------------------------------------------
| Version |                                                  |
-------------------------------------------------------------
| Author |                                                   |
-------------------------------------------------------------
| Distribution |                                             |
-------------------------------------------------------------
| Disclaimer |                                               |
-------------------------------------------------------------
| Contact |                                                  |
-------------------------------------------------------------
| Document |                                      (MORE?) |
-------------------------------------------------------------
|   Section |                                     (MORE?) |
-------------------------------------------------------------
|      Topic |                                    (MORE?) |
-------------------------------------------------------------
| INTRODUC |                                    | Unit       |
-------------------------------------------------------------
| USAGEEXA |                                    | Unit       |
-------------------------------------------------------------
| LISTOFCO |                                    | Unit       |
-------------------------------------------------------------
```

Figure 21. User Manual (USERMAN) Unit Class

Figure 22 shows the user view of the Unit created from the USERMAN Unit Class. The structure is of course the same, but the replication of the topic categories is shown. Only one line of text appears for each topic, because the sections are quite long. The trailing dots (....) means that more text follows. Note also that the Entries referring to the Unit Classes, INTRODUC, USAGEEX and LISTOFCO are shown as being refined. This is indicated by the title text shown in the center and the existence of the Unit serial number at the far right of the Entry.

```
-------------------------------------------------------------
| TRIADMD-USERMAN | Notes for Method Designer   | UNIT 19   |
-------------------------------------------------------------
| Version | 3 dated 1 1 86                                  |
-------------------------------------------------------------
| Author | Hochstettler and Ramanathan                      |
-------------------------------------------------------------
| Distribution | Upon request                               |
-------------------------------------------------------------
| Disclaimer |                                              |
-------------------------------------------------------------
| Contact | Dr. Jay Ramanathan                              |
-------------------------------------------------------------
| Document | How to use TRIAD                      (MORE?) |
-------------------------------------------------------------
|   Section | Getting Started                      (MORE?) |
-------------------------------------------------------------
|     Topic | Getting Started                      (MORE?) |
  TRIAD is a shell environment .......
-------------------------------------------------------------
|     Topic | Defining the method on paper         (MORE?) |
  Problem solving methods divide ......
-------------------------------------------------------------
|     Topic | Using the TRIAD Tuner                (MORE?) |
  After logging into the IBM system ......
-------------------------------------------------------------
|     Topic | Attributes                           (MORE?) |
  Predefining attributes .......
-------------------------------------------------------------
| INTRODUC | Terminology and Guided Tour    | Unit 20   |
-------------------------------------------------------------
| USAGEEXA | Usage Example                  | Unit 29   |
-------------------------------------------------------------
| LISTOFCO | TRIAD Commands                 | Unit 30   |
-------------------------------------------------------------
```

Figure 22. Completed User Manual (USERMAN) Unit

Figure 23 shows a part of the list of Units created from the TRIAD Method. The numbers to the left of the Unit Class name indicates the level of the Unit. For instance the unit PROJECT is the Initial Unit and is assigned a level of 1. All Units refined from the PROJECT Unit have a level of 2 and so on. The levels are also indented to impart a visual image of the levels to the user.

This list serves several purposes, much like the table of contents of a book. First it shows each available unit and gives the serial number, which allows a user to display it directly without navigating through the network of Units. Second it summarizes the Units' contents by showing the Unit Class name, its title and all units refined from it. Thus it shows the structure of the method in an outline form.

7.2.2 PROGRAM STRUCTURE METHOD

Although Figure 23 is only a partial list of all of the Units created from the TRIAD Method Unit Classes, it shows most of the Units and imparts the structure of the multiple methods applied in the TRIAD Method. The program structure method is shown at the bottom of the figure. Unfortunately the TRIAD Tuner program structure, as shown in Figure 24 does not have an interesting structure, since each operator has a module that is invoked by a command name. Thus, the expressive power of the method is not directly shown. From the method definition of the Program Structure method it can be seen that complex software structures can be represented with these Unit Classes. The major component Tuner, shown in the MAJORCOM Unit of the figure has three Units refined from it. The REVIEW Unit describes the experience with a Tuner prototype. The data structure Units describe the major data structures used by the tuner routines. Finally for each routine a MODULES Unit is listed. Each MODULES Unit describes the implementation of the command in detail. It also has a pointer to the file containing the source code for review or modification.

From these figures it should be noted that the TRIAD Method attempts to localize information about a concept. For example, the Program Structure method uses a two tier structure to organize a program. All of the modules are owned by the major component. In addition the data structure descriptions are localized with the major component.

1. PROJECT	TRIAD Software Engineering Env.	Unit 1
2. MEMBER	Dr. Jay Ramanathan	Unit 2
2. MEMBER	Mr. Thorbjorn Andersson	Unit 3
2. MEMBER	Mr. William H. Hochstettler,III	Unit 4
2. MEMBER	Mr. Ronnie Sarkar	Unit 5
2. MEMBER	Mr. Robert Vermilyer	Unit 6
2. MEMBER	Mr. Ronald Hartung	Unit 7
2. MEMBER	Mr. James Davenport	Unit 8
2. USERSMAN	Notes for the method designer	Unit 19
3. INTRODUC	Terminology and guided tour	Unit 20
4. TERM	Method	Unit 21
4. TERM	Unit Class	Unit 21
4. TERM	Blank Unit	Unit 22
4. TERM	Unit List	Unit 24
4. TERM	Tuning	Unit 25
4. TERM	Instantiating	Unit 26
4. TERM	Refining	Unit 27
3. USAGEEX	Usage Example	Unit 29
3. LISTOFCO	TRIAD Commands	Unit 30
2. PHASEO	Project Objectives	Unit 31
2. PHASEI	Overall Architecture	Unit 32
3. REVIEW	Suggestions for Product Arch.	Unit 309
3. FUNCTION	Functional Overview	Unit 33
3. CONFIGUR	Configuration Specifications	Unit 34
2. PHASEII	Programming Logic	Unit 35
3. HISTORY	Reason for the SIL breakdown	Unit 389
3.MAJORCOM	Tuner component	Unit 36
4. REVIEW	Experience with prototype	Unit 308
4. DATASTRU	Method Lists	Unit 86
4. DATASTRU	Blank Units	Unit 87
4. DATASTRU	Attributes	Unit 88
4. MODULES	AddAttribute	Unit 70
4. MODULES	DeleteAttribute	Unit 244
4. MODULES	DeleteUnit	Unit 245
4. MODULES	DeleteCategory	Unit 72
4. MODULES	NewCategory	Unit 93
4. MODULES	NewUnit	Unit 78
4. MODULES	PrintBUL	Unit 70
4. MODULES	Retitle	Unit 74

Figure 23. Partial List of Units from the TRIAD Method

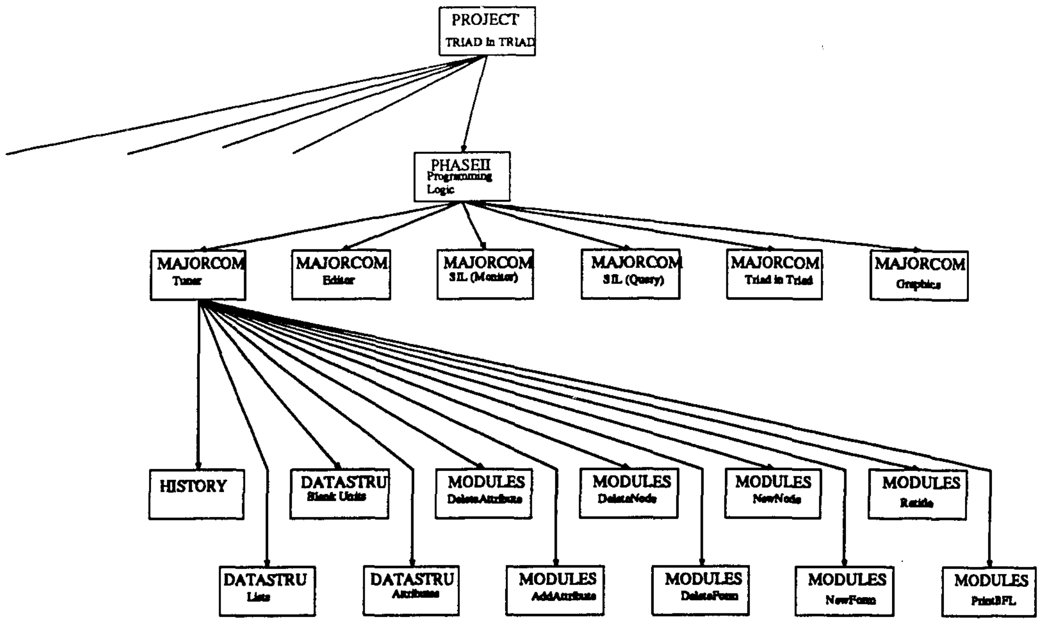


Figure 24. Structure of Units for the Tuner Major Component

CHAPTER VIII

TRIAD MODEL EVALUATION

The evaluation of any model is best done by determining how well the model actually reflects the object being modeled. This chapter reviews the features of the TRIAD model and their applicability to software engineering methods. Software engineering methods are of two general forms; either textual or representational. Textual methods merely organize large text collections for convenient use and comprehension. Representational methods attempt to model problem solutions or software by using compact notations, usually graphs.

The TRIAD model models textual software engineering methods extremely well. The model supports the storage of text in its original format as a text type Attribute. In addition, the text can be partitioned into Unit Classes and Component Categories within each Unit Class. This feature allows the text to be subdivided to manageable pieces. Also all of the Attributes can be applied to the individual pieces of text thereby increasing the power and meaning of the Attributes by specification. The Refinement Linkages allow a block of text to be refined into more specific concepts, creating the ability to organize large blocks of text into a network of smaller related pieces. The Attributes allow descriptive values about the text to be stored separate from, but physically adjacent to the text they modify. Additional support is provided by the TRIAD editor which allows the text to be edited directly within the Component Entry. Further, the Procedures allow procedural knowledge about the text to be associated with Component Categories and Unit Classes, thereby offering the software engineer help in using the method. Secondary links from one Entry to another Entry allows the expression of a relationship that is different from that of the Refinement Linkage (ownership).

Support for representational types of software engineering methods, such as SADT, data flow diagrams or flowcharts is similar to textual method support except for the meaning of the Refinement Linkage and the user view of the method. The representational methods create diagrams of software and to be effectively supported by a software engineering environment, these pictures must be represented and displayed. Although TRIAD depends heavily upon the graphical interface to draw and edit the pictures, the TRIAD Model has a structure that allows a direct translation from the model to a graphic representation.

The Dataflow Diagram example presented in Chapter II illustrates the features of the TRIAD model that represent graphical methods. The Refinement Linkage is used to represent the arcs in the graphical methods. The Component Category contains an Attribute to store the text usually contained within the nodes of the graph or attached to the arc. The icon Attribute associated with the Unit Classes, allows the method designer to design and name an icon independent of its use in the method. These three features of the TRIAD Model make it very easy to represent directed graph based methods. Using this representation, a graphics interface can display the graphical representation of the software which the user can view and manipulate.

Further, the Procedures provide the same capabilities for graphical software engineering methods as for textual software engineering methods, namely to encode procedural knowledge about the software engineering method and its application.

The Procedures in addition to providing the means for encoding the rules and policies of a method are used to create extended commands and build interfaces to existing tools. The Procedures are implemented using a procedural language provided by the TRIAD model implementation. Also provided by the implementation are primitives for navigation through the network of Units and manipulation of the TRIAD model elements. These facilities allow extended commands to be built, which accomplish tasks specific to a method and even more specific to the software being implemented. For example, a Procedure can be written to navigate through a call structure method and collect the percentage completed of the coding of each module. The collected percentages can be combined to represent a project completion percentage. This type of processing is likely to be repeated periodically by a project manager to evaluate the current progress of the project. By creating a Procedure, giving it a distinct command name and then putting references to the command name in the places in the method where the manager is likely to request the information generated by the command, the user is assisted in his job.

Tool interfaces are also implemented using the Procedures because the navigation and extraction primitives provide the means of placing information into a format acceptable to tools. The text formatter example in Chapter VI demonstrates the power that the tool interface provides to the user to exploit existing products. The significant point of the text formatter interface is that most of the interface was done by using the features already provided by the TRIAD model. No additional "fudging" was required. The formatter Attributes were created using the Attribute facility provided. The text extraction primitives were used to get the text from the Entries in the Entries and then the formatter was invoked.

8.1 RESEARCH CONTRIBUTIONS

This dissertation described a model for representing multiple software engineering methods in a software life cycle. Due to the number and difference in software engineering methods for the various phases of the software life cycle, this model provides a general representation for identifying the basic elements in most methods. Further, computer support can be provided to models that previously were unsupported and to new methods not yet defined, by describing the method using the model and by using the computer support package provided with the implementation of the model.

The model was implemented to demonstrate that the model specification was capable of being implemented. The implementation was used to evaluate the model and gain insight into further extensions and enhancements to the model. By expressing several software engineering methods in the model, implementation experience with multiple methods was also gained.

8.2 FUTURE ENHANCEMENTS

The ability of the TRIAD Model to represent methods is clear both by analysis and the application of the model to various methods. Future work in the support of methods is along the following divergent lines:

- o Knowledge based support of software engineering methods,
- o Specialization of TRIAD Model elements to support classes of methods and
- o Integration of operating system and database concepts into the implementation to improve performance of the prototype.

Knowledge based support of software engineering methods will focus more AI and expert systems techniques on software engineering tasks. This work can proceed from a solid base of the TRIAD Model, which can be used to represent information and knowledge. The application of AI techniques will still be in the assisting role rather than one of automatic programming. Simple applications of AI are possible by using Procedures to implement local procedures representing expert knowledge about design and coding. Open questions still remain as to the best way of building these procedures. The usual technique of using a general purpose procedural language may not be as good as a declarative language.

The current approach of writing Procedures using implementation provided primitives within a procedural programming language provides the software engineer sufficient power, but not much help in applying a method. Advances in languages to assist in the creation of these Procedures will increase the ability of method users to utilize the power of the model without an investment in time and effort similar to writing programs. The creation of such a language implies that a greater understanding of the requirements of such Procedures is available. At this time, experience with the model and its implementation has not produced sufficient knowledge to design a higher level Procedure language. However, as experience with the model is gained, insight on the use of the Procedures may provide the means for designing a easier to use Procedure implementation language.

During the creation and application of the software engineering methods to support the TRIAD development, it was clear that certain Attributes were necessary to support the methods. As more features were added, more special Attributes were required. This trend is analogous to database research where general models have been modified and extended to support a specific class of problems with built-in types [SU86]. This same process of specializing will continue both in the software engineering method domain as more methods are applied using TRIAD, and also as new problem domains are explored.

Finally, the TRIAD implementation as a prototype, adequately demonstrated the concepts of software engineering environments to support multiple software engineering methods. To learn more about the support a software engineer needs on the job, a more responsive implementation is required. The capabilities of the implementation need to be increased by ensuring data integrity and allowing multi-user access. Although these are primarily database implementation issues, the use of IBM's SQL demonstrated that the loss of performance is not necessarily offset by a gain in power and function. TRIAD has many of the aspects of a database (data model and query language), therefore, TRIAD needs to use the physical level access techniques of a database system to improve its performance. If TRIAD's performance can be improved, it will become a laboratory for studying the definition and use of software engineering methods in particular and environments in general.

LIST OF REFERENCES

ALFO85 ALFORD, MACK, "SREM at the Age of Eight: The Distributed Computing Design System", IEEE Computer, Volume 18, Number 4, (Mar 1985) p. 36-46.

AMBR84 AMBRIOLA, V., GAIL E. KAISER AND ROBERT J. ELLISON, "An Action Routine Model for ALOE", Tech. Report, Dept. of CS, CMU, August, 1984.

BERG79 BERGLAND, G.D., "Structured Design Methodologies", Tutorial: Software Design Strategies G.D. Bergland editor, IEEE, Long Beach, Ca., 1979, p. 162-181.

BIGG80 BIGGS, CHARLES L., AND WILLIAM ATKINS, Managing the Systems Development Process, Prentice-Hall, Englewood NJ, 1980.

BOEH84 BOEHM, BARRY W. AND ET AL, "A Software Development Environment for Improving Productivity", IEEE Computer, Volume 17, Number 6, (June 1984), p. 30-44.

BORG85 BORGIDA, ALEXANDER, "Features of Languages for the Development of Information Systems at the Conceptual Level", IEEE Software Volume 2, Number 1, (January 1985), p. 63-72.

BROD84 BRODIE, MICHAEL L., "On the Development of Data Models", On Conceptual Modeling, Brodie, Mylopoulos, Scmidt, eds., Springer-Verlag, 1984, Chapter 2, p. 19-47.

CAIN77 CAINE, STEPHEN H. AND E. KENT GORDON, "PDL--A Tool for Software Design", Tutorial on Software Design Tools, IEEE, Long Beach, Ca., 1977, p. 168-173.

CHEN76 CHEN, PETER, "The Entity-Relationship Model--Towards a Unified View of Data", ACM Transactions on Database Systems, Volume 1, Number 1, (March 1976) p. 9-36.

CHIK85 CHIKOFSKY, ELLIOT AND DANIEL TEICHROEW, "Generating Flexible Methodology-Specific System Development Environments", The Proceedings of the ACM-IEEE SOFTFAIR II, (December 1985), p. 24-31.

DAHL72 DAHL, O-A, E. W. DIJKSTRA AND C. A. R. HOARE, Structured Programming, Academic Press, New York, NY, 1972, 220 pages.

DATE77 DATE, C.J., An Introduction to Database Systems, Addison-Wesley, Second Edition, 1977.

DAVE86 DAVENPORT, JAMES, The Use of a Relational Database on a Software Engineering Environment, Masters Th., in progress, The Ohio State University, Columbus, Ohio, December 1986.

DAVI83 DAVIS, WILLIAM S., Tools and Techniques for Structured Systems Analysis and Design, Addison-Wesley, Reading, Ma, 1983, 187 pages.

DEMA79 DEMARCO, TOM, Structured Analysis and System Specification", Prentice Hall, New York, NY, 1978, 339 pages.

DEME82 DEMETROVICS, JANOS, ELOD KNUTH AND PETER RADO, "Specification Meta Systems", IEEE Computer, Volume 15, Number 5, May 1982, p. 29-35.

DOLO78 DOLOTTA, T.A., R.C. HAIGHT AND J.R. MASHEY, "The Programmer's Workbench", The Bell System Technical Journal Volume 57, Number 6, (July-August 1978), p. 2177-2200.

FIKE85 FIKES,RICHARD AND TOM KEHLER, "The Role of Frame-Based Representation in Reasoning", Communications of the ACM, Volume 28, Number 9, Sept. 1985, p. 904-920.

FREE77 FREEMAN, PETER AND ANTHONY I. WASSERMAN (EDS.), Tutorial on Software Design Techniques, 2nd Ed., IEEE, Long Beach, Ca., 1977, 288 pages. Number ,

HAMM81 HAMMER, MICHAEL, "Database Description with SDM: A Semantic Database Model", ACM Transactions on Database Systems, Volume 6, Number 3, (Sept. 1981) p. 351-386.

HART87 HARTUNG, RONALD, The design and Application of Graphics for TRIAD, Ph.D. Th., in progress, The Ohio State University, Columbus, Ohio, March 1987.

HEAC79 HEACOX, H. C., "RDL: A Language for Software Development", ACM SIGPLAN Notices Volume 14, Number 12, December 1979, p. 71-79.

IBMD IBM, Document Composition Facility: Users Guide, 4th edition, IBM Corporation, New York, 1983, 415 pages.

IBMG IBM, GDDM Reference Manual, IBM Corporation, New York, 1983, 321 pages.

IBMR IBM, System Product Interpreter Reference, First Edition, IBM Corporation, New York, 1983, 160 pages.

IBMX IBM, System Product Editor Reference, edition, IBM Corporation, New York, 1982, 78 pages.

JACK78 JACKSON, MICHAEL A., Principles of Program Design, Academic Press, New York, NY, 1975, 297 pages.

KENT79 KENT, WILLIAM, "Limitations of Record-Based Information Models", ACM Transactions on Database Systems, Volume 4, Number 1, (March 1979), p. 107-131.

KNUT68 KNUTH, D. E., "Semantics of Context-free Languages", Mathematical Systems Theory Volume 2, Number 1, (1968), p. 127 145.

KUO83 KUO, J. C., Design and Implementation of a Form-Based Software Environment, Ph.D. Th., The Ohio State University, Columbus, Ohio, August 1983, 328 pages.

LAUB82 LAUBER, RUDOLF, "Development Support Systems", IEEE Computer Volume 15, Number 5, May 1982, pp. 36-46.

MATH86 MATHIS, ROBERT, F., "The Last 10 Percent", IEEE Transactions on Software Engineering, Volume 12, Number 6, June, 1986, p. 705-712.

MCKN85 MCKNIGHT, WALTER L., <u>A Meta System for Generating Software Engineering Environments,</u> Ph.D. Th., The Ohio State University, Columbus, Ohio, June 1985, 277 pages.

MINS75 MINSKY, MARVIN, "A Framework for Representing Knowledge", <u>The Psychology of Computer Vision,</u> Patrick Henry Winston, Ed., McGraw-Hill, 1975, p. 211-277.

PARK78 PARKER, JOHN A., "A Comparison of Design Methodologies", <u>ACM SigSoft Software Engineering</u> Notes, Volume 3, Number 9, October 1978.

PARN79 PARNAS, DAVID L., "On the Criteria to be used in Decomposing Systems Into Modules", <u>Communications of the ACM,</u> Volume 13, Number 12, (Dec. 1972), p. 131-136.

PARN85 PARNAS, DAVID L., PAUL C. CLEMENTS AND DAVID M. WEISS, "The Modular Structure of Complex Systems", <u>IEEE Transactions on Software Engineering,</u> Volume 11, Number 3, (Mar. 1985), p. 259-266.

PETE79 PETERS, LAWRENCE J. AND LEONARD L. TRIPP, "Comparing Software Design Strategies", <u>Tutorial: Software Design Strategies,</u> Bergland 243 Gordon (eds.), IEEE Long Beach, Ca., 1979, p. 185-188.

POLA78 POLAN, MARVIN AND FRED J. SARISONEN, <u>Software Design Methodology Interim Report,</u> Teledyne Brown Engineering, Huntsville, Al., May 1978.

PYE69 PYE, DAVID, <u>The Nature of Design,</u> Studio Vistal/Reinhold, New York, NY, 1975.

RAMA86 RAMAMOORTHY, C.V., V. GARG AND A. PRAKESH, "Programming In The Large, <u>IEEE Transactions on Software Engineering,</u> Volume 12, Number 7, July, 1986 p. 769-783.

RICH83 RICH, ELAINE, <u>Artificial Intelligence,</u> McGraw-Hill, New York, NY, 1983, 436 pages.

ROSS77a ROSS, DOUGLAS T. AND KENNETH E. SCHOMAN,JR., "Structured Analysis for Requirements Definition", <u>IEEE Transactions on Software Engineering,</u> Volume 3, Number 1, (Jan. 1977), p. 6-15.

ROSS77b ROSS, DOUGLAS T., "Structured Analysis(SA): A Language for Communicating Ideas", IEEE Transactions on Software Engineering, Volume 3, Number 1, (Jan. 1977), p. 16-33.

ROSS85 ROSS, DOUGLAS T. "Interview: Douglas Ross Talks about Structured Analysis", IEEE Computer Volume 18, Number 7, (July 1985), p. 80-88.

RUBI85 RUBIN, HOWARD A., D.C. VON KLEECK AND DAVID BARTZ, "Integrating Software Development Estimation, Planning, Scheduling and Tracking: The PLANMACS System", Proceedings of the ACM-IEEE SOFTAIR II, (Dec. 1985), p. 24-31.

SCHE78 SCHETHTER, DAVID, "The Skeleton Program Methodology", Datamation November, 1978, p. 147-150.

SOFT81 SOFTECH, INC., "Integrated Computer-Aided Manufacturing (ICAM)", Tech Report, Materials Laboratory, Wright-Patterson AFB, June, 1981, 141 pages.

SONI83 SONI, D. A., Design and Modeling of TRIAD - An Adaptable, Integrated Software Environment, Environment, Ph.D. Th., The Ohio State University, Columbus, Ohio, June 1983, 247 pages.

STAY77 STAY, J. F. "HIPO and Integrated Program Design", Tutorial on Software Design Techniques, Freeman and Wasserman (eds.), IEEE Long Beach, Ca., 1977, p. 174-178.

STEV77 STEVENS, W. P., G. J. MYERS AND L. L. CONSTANTINE, "Structured Design", Tutorial on Software Design Techniques, Freeman and Wasserman (eds.), IEEE Long Beach, Ca., 1977, p. 97-100.

SU86 SU, STANLEY Y. W., "Modeling Integrated Manufacturing Data with SAM*", IEEE Computer, Volume 19, Number 1, January 1986, p. 34-49.

TEIC77 TEICHROEW, DAVID AND ERNEST A. HERSHEY,III, "PSL/PSA: A computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems", IEEE Transactions on Software Engineering, Volume 3, Number 1, (Jan. 1977) p. 41-48.

TSIC82 TSICHRITZIS, DIONYSIS AND FREDERICK H. LOCHOVSKY, Data Models, Prentice-Hall, 1982, 381 pages.

WINO72 WINOGRAD, TERRY "Frame Representations and the Declarative/procedural Controversy", Representation and Understanding, Bobrow and Collins (eds.), Academic Press, 1975, p. 185-210.

YAU86 YAU, S.S. AND J. J-P. TSAI, "A Survey of Software Design Techniques", IEEE Transactions on Software Engineering, Volume 12, Number 6, July, 1986, p. 703-721.

YOUR78 YOURDON, EDWARD AND LARRY L. CONSTANTINE, Structured Design, Second Ed., Yourdon Press, New York, 1978.

YOUR86 YOURDON, EDWARD "What Ever Happened to Structured Analysis", Datamation, Volume 32, Number 11, June 1986, p. 133-138.