# A computer aided software engineering tool specification for the scenario-based engineering process

Samad, Ashequs, M.S.

The University of Texas at Arlington, 1993

# A COMPUTER AIDED SOFTWARE ENGINEERING TOOL SPECIFICATION FOR THE SCENARIO-BASED ENGINEERING PROCESS

The members of the Committee approve the masters thesis of Ashequs Samad

| | |
|---|---|
| Stephen P. Hufnagel<br>Supervising Professor | Stephen P Hufnagel |
| Karan A. Harbison | [illegible] |
| Diane J. Cook | Diane J. Cook |

# A COMPUTER AIDED SOFTWARE ENGINEERING TOOL SPECIFICATION FOR THE SCENARIO-BASED ENGINEERING PROCESS

by

ASHEQUS SAMAD

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT ARLINGTON

December 1993

## ACKNOWLEDGEMENTS

I wish to convey my sincerest appreciation and gratitude to Dr. Stephen P. Hufnagel for supporting me in conducting my research. I thank him for giving me guidance and insight regarding theoretical concepts as well as in practical issues. This work would not be possible without the resources made available by all my committee members. I also wish to thank Dr. Karan A. Harbison and Dr. Diane J. Cook for giving me inspiration, encouragement and support.

July 22, 1993

## ABSTRACT

# A COMPUTER AIDED SOFTWARE ENGINEERING TOOL SPECIFICATION FOR THE SCENARIO-BASED ENGINEERING PROCESS

Publication No. _____

Ashequs Samad, M.S.
The University of Texas at Arlington, 1993

Supervising Professor: Stephen P. Hufnagel

This work specifies Computer Aided Software Engineering (CASE) tool principles and requirements to support the Scenario-based Engineering Process (SEP). An eventual goal is to develop a domain specific software architecture (DSSA) for CASE tools. The environment specified should provide a collection of tools which support vital software engineering activities covering the entire development life cycle as well as those within specific phases.

The first part provides a qualitative analysis of the following tools: Teamwork/OOD, Teamwork/HOOD, Teamwork/Ada, OOATool, C++ Designer and OpenSELECT. Critical software development criteria are the basis of this evaluation. The second part defines the CASE environment DSSA. A notation for the process and the software configuration items are identified. The use of the tools in each phase and an architectural framework for integration of the tools is presented. Design specifications using the SEP are also developed.

## TABLE OF CONTENTS

## LIST OF ILLUSTRATIONS

## LIST OF TABLES

# 1 INTRODUCTION

A paradigm shift to the software factory of the future, envisioned as an environment which provides the basis and means of software development utilizing and maximizing the strengths while minimizing the weaknesses of both the developer and tools, is the direction of evolution in software engineering [Fernstrom, 92]. An integrated, unfragmented development environment supporting seamless movements from one phase to another at an acceptable and measurable risk level is the defining feature of the effective software factory. The competitive edge in such an environment comes from easy to use and standardized components, which are shared effectively, and from the automatic management of documentation and configuration complexity of large systems under development. Thus, the higher capital investment in tools can sufficiently be justified and accepted in practice.

This work targets the establishment of a project knowledge base and an integrated software construction framework of software tools to build, maintain, and reuse quality software. The key innovative concepts are the scenario-based responsibility-driven processes and a refined and robust object oriented approach with which to formally specify, develop, assess, and retrieve components. The main goal is to allow and maintain overall conceptual integrity and simplicity, while bringing together diverse and evolving technologies and concepts into a unified environment.

This work uses as foundation the experiences and observations during development efforts using six different CASE tools. Three of these tools are part of the Teamwork set of tools by Cadre Technologies, while the others have been developed by leading proponents of object oriented methods. Small models were developed individually and a large real-time system was specified by a team of developers working in groups. This was done using each tool. The first part of this work is an evaluation based on this experience. The purpose of this evaluation is 1) to identify strengths, weaknesses and important factors, and 2) to learn from the evaluation to be able to propose a robust method or tool. The evaluation is qualitative and it addresses important and critical issues in software engineering. The assessment also provides insight into such properties of the tool, as its assumed set of software configuration items and method of use.

The scenario-based engineering process is the key concept for the next part of the thesis. The SEP is the first instantiation of the Domain Specific Software Architectures (DSSA) philosophy. The life cycle for this process and its fundamental concepts have been described by Haddock et al [Haddock, 93]. A language independent, object oriented notation is selected to support analysis and design phases of the responsibility driven, scenario-based engineering process and its life cycle. The proposed or selected notation attempts to combine and use well known abstraction mechanisms from the literature on both object oriented and functional development methods. The representation of scenarios is a key concept of the notation.

The next portion of the thesis concentrates on identifying system configuration items and models, describing a development process with the use of a proposed tool, and developing a specification of an integrated tool environment. The SCIs are composed of models using the defined notation and the development process addresses their role in each phase of the scenario-based engineering process life cycle. The tool specification consists of the tool architecture, which uses a layered approach in order to identify and specify major components, their organization and integration related issues. An object model and a scenario model which represents key activities are also developed.

## 2 COMPARATIVE STUDY OF OBJECT ORIENTED TOOLS

A number of object oriented methodologies have been proposed in the literature. Recent studies elaborate on the features, strengths, and weaknesses of these methods [Korson, 90]. Studies comparing and contrasting these methods and pointing out relevant differences [Champeaux, 92] are also available. One important aspect of the proposed methods is the CASE (Computer Aided Software Engineering) tool support available for each method. Proponents and authors of most of the methods have successfully made CASE tool support for their methods available in the market. This section is an evaluation of several methods and their associated tools.

**Table 1. Basic Information of Tools Surveyed**

| Tool | Vendor | Version/ Release | Release Date | Platforms | Additional Software |
|---|---|---|---|---|---|
| Teamwork/HOOD | Cadre Technologies Inc. | 4.1 | 1992 | Unix | X Windows |
| Teamwork/Ada | Cadre Technologies Inc. | 4.0 | 1990 | Unix | X Windows |
| Teamwork/OOD | Cadre Technologies Inc. | 4.1 | 1992 | Unix | X Windows |
| OOATool | Object International Inc. | 1.3.1 | July 31 1992 | DOS | MS Windows |
| C++ Designer | Select Software Tools Ltd. | 3.0 | 1993 | DOS | MS Windows |
| OpenSELECT | Select Software Tools Ltd. / Meridian | 3.01 | 1991 | DOS | MS Windows |

The evaluation begins with a taxonomy that captures information like the tool name and vendor, release date, life-cycles, phases and methods supported, platforms in which the tools operate, etc. The tools evaluated are introduced in Table 1 and Table 2 provides information about the methods supported, proponents and the basic functions supported by the tool. However, the list of functions does not provide complete information regarding the tools functions but attempts to classify the tools broadly

**Table 2. Methods supported by tool and tool functions**

| Tool | Method/Technique | Author/ Proponent | Basic Functions |
|---|---|---|---|
| Teamwork/HOOD | Hierarchical Object Oriented Design (HOOD) | European Space Agency (ESA) | Requirements analysis, configuration mgmt., design and documentation |
| Teamwork/Ada | Ada Structure Graphs (ASG) | R.J.A. Buhr | Requirements analysis, configuration mgmt., design, code generation and reverse engineering |
| Teamwork/OOD | Object Oriented Design Language (OODLE) | Sally Shlaer & Stephen J. Mellor | Requirements analysis, configuration mgmt., design, documentation and code generation |
| OOATool | Object Oriented Analysis | Peter Coad & Ed Yourdon | Requirements analysis, configuration mgmt., design, documentation, reuse and code generation |
| C++ Designer | Object Modelling Technique | James Rumbaugh et al | Requirements analysis, configuration mgmt., design, documentation, reuse and code generation |
| OpenSELECT | Hierarchical Object Oriented Design (HOOD) | European Space Agency (ESA) | Requirements analysis, configuration mgmt., design and documentation |

The general approach to the assessment is qualitative. The approach presented by Mosley [Mosley, 1992] for a quantitative assessment is modified to suit this purpose. The goal is to identify the purpose(s) of the tool and to see if the tool sufficiently and adequately supports the entire development life cycle or the targeted phase(s).

The first part of the evaluation is based on studying the robustness of the analysis/ design method which the tool supports. In order to do this an overview of object-oriented concepts is presented with relevant references to each tool. Although most of the methods studied converge on modelling abstractions and their use, and agree about the major needs in terms of system representation, actual experience with the tools in development of large, complex systems reveals issues that are critical to the success and usefulness of the tool. The purpose of this evaluation is to explore these issues. The evaluation attempts to identify primary features, commonalities, major differences, strengths and weaknesses.

The second part of the evaluation presents various issues relevant to the development method, life cycle and tools. Each issue presented can also be seen as criteria for evaluation of the tool. The issues and items discussed are development life-cycle, method and graphic notation, scope of the tool, its products, architecture and how the tools support various vital software engineering activities such as re-engineering, reverse engineering, reuse and configuration management.

The final part of the evaluation provides a summarized evaluation for each tool by extracting the critical and essential details about the tool. These discussions are tailored to each tool in order to provide insight regarding the tool on the basis of the criteria presented in the first two parts. Specific issues regarding each tool and its organization are also presented in these discussions.

## 2.1 Overview of Object Oriented Concepts

In this section the basic and most important concepts and features of object oriented software development are discussed. Most of the discussed features are common among all methods proposed. However, certain features that are not common to all are also discussed due to their usefulness and practicality, and are considered to be contributions to the paradigm. Table 3. summarizes the capability of each tool to support the following concepts and abstractions.

### 2.1.1 Objects and Classes

The concept of an object is similar in all the methods. Objects are conceptual entities that refer to an identifiable and tangible thing or construct in the domain or system under consideration [Coad, 91]. The notation of an object represents the definition of the interface of the object. Most methods do not require the specification of certain parts of the object interface such as types and constants. Objects allow and encourage principles of data abstraction rather than procedural abstraction. Their characteristics such as unique identity, state and behaviour, and contents such as attributes and services or operations are also largely similar in all the methods.

A class is a meaningful grouping of objects, where each object is considered to be an instance of the class to which it belongs. This concept is explicitly addressed in OOA [Coad, 91] and notational difference between the two is supported by the tool. All the other methods use classes and objects interchangeably, or do not support the class abstraction at all.

#### 2.1.1.1 Passive Objects

Passive objects provide purely sequential control flow within the object, and are therefore considered to have no control over execution. Calling a service of a passive object leads to immediate execution of the service or function, and upon completion control is transferred back to the calling procedure. Objects are classified as active, passive, agent, etc. in HOOD [HOOD Working Group, 89] and by Booch [Booch, 90].

#### 2.1.1.2 Active Objects

An active object has its own control flow. The HOOD Reference Manual [HOOD Working Group, 89] states "that when an operation or service of an active object is called, control is not necessarily transferred to the operation immediately, but the active object receives an external stimulus. Reaction to the stimulus may be delayed according to the internal state of the object, or based on the type of execution request". This leads to classification of different types of constraints defined on each operation or service. HOOD requires an active object to contain at least one constrained service

### 2.1.2 Features

Features of an object include attributes and services [Meyer, 88]. Each object is described by its data and the services operating on the data. However, HOOD and Ada Structure Graphs [Buhr, 84] do not have mechanisms of explicitly representing attributes of objects. Data flow among operations are identified in these methods, thus alleviating this problem. Attributes and services can be defined explicitly in terms of their data types, although only OODLE requires the specification of data types. Authors use the terms "method", "operation", and "service" interchangeably.

### 2.1.3 Polymorphism

Polymorphism is a property of objects. Same service names may exist in multiple object types. Although the action taken by each occurrence of an operation may be similar, the implementation and end result will vary greatly depending on the object type. Such operations or services can be considered polymorphic. An object may also contain multiple implementations for the same service or operation. Polymorphism within an object need not be resolved during analysis but must be addressed in design documents from which code is generated.

### 2.1.4 Encapsulation and Information Hiding

Information hiding suggests that each component of a system representation should encapsulate details of implementation that are unnecessary to the components use. Object oriented programming languages achieve this by not allowing direct access to the data of each object. The data can only be operated upon through a service provided by the object. The interface to each model is defined in such a way that internal details are not revealed. Defining objects in terms of their attributes and services achieves this as functions and data required for internal operations are not shown. HOOD [HOOD Working Group, 89], ASG [Buhr, 84], and OODLE [Shlaer, 90] provide a greater detail of internal operations

**Table 3. Object-Oriented concepts or abstractions supported by tool**

| Tools | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Teamwork/HOOD | Y |  | Y | Y |  | Y | Y |  |  |  | Y |  |
| Teamwork/Ada | Y |  | Y | Y | Y | Y |  |  |  |  | Y |  |
| Teamwork/OOD | Y |  |  |  | Y | Y | Y | Y | Y |  | Y |  |
| OOATool | Y | Y |  |  | Y | Y | Y | Y | Y | Y | Y | Y |
| C++ Designer | Y |  |  |  | Y | Y | Y | Y | Y | Y |  |  |
| OpenSELECT | Y |  | Y | Y |  | Y | Y |  |  |  | Y |  |

The numbers at the heads of the columns Table. 3 correspond to the following Object-Oriented concepts:

| | |
|---|---|
| 1 = | objects? |
| 2 = | classes are distinguished? |
| 3 = | passive objects? |
| 4 = | active objects? |
| 5 = | attributes? |
| 6 = | operations or services? |
| 7 = | aggregation or assembly? |
| 8 = | generalization-specialization? |
| 9 = | multiple inheritance? |
| 10 = | instance connections or associations? |
| 11 = | message connections? |
| 12 = | can be used to support scenarios? |

### 2.1.5 Structures, Aggregation and Classification

Object structure representation is usually in two forms: aggregation or assembly and classification or generalization-specialization. The former indicates that a set of objects are combined to form another object. This relationship between objects is usually of the type "is a part of" or "whole-part". The latter indicates a specialized object derived from a generalized object. This relationship between objects is of the type "is a". In this case the specialized objects may inherit or redefine the properties of the generalized object. Multiple inheritance refers to the situation where an object is a specialized form of two or more objects, thus inheriting characteristics from both. Rumbaugh et al [Rumbaugh, 91] propose several methods of resolving the conflicts arising from this situation.

Aggregation or assembly structures are implicitly a hierarchical abstraction technique, and are available in all object oriented methodologies. All of the tools exploit this hierarchical abstraction technique explicitly. The concept of classification is not supported by Ada Structure Graphs [Buhr, 84] or HOOD [Hood Working Group, 89]. In all the other tools classification is used to show inheritance, where objects can be related as superclasses and subclasses.

#### 2.1.5.1 Inheritance

The concept of inheritance applies to generalization-specialization relationship, used to derive objects. The specialized object selectively inherits functions and attributes of the generalized object. The attributes and services do not need to be shown graphically within the specialized object. Their existence is assumed. Only additional services and attributes need to be shown. Inheritance is supported by the OOA [Coad, 91], OMT [Rumbaugh, 91] and OODLE [Shlaer, 90] as these methods support classification. A drawback of these tools is the inability to collapse an inheritance structure into an instantiated process object structure.

#### 2.1.5.2 Multiple Inheritance

Multiple inheritance arises when an object is specialized from two or more other objects. This induces an ambiguity and possible conflicts which the analyst needs to resolve. Both C++ Designer and OOATool allow for multiple inheritance and their consistency checking or critiquing feature identifies attributes or services which conflict.

### 2.1.6 Instance Connections and Associations

OMT [Rumbaugh, 91] and OOA [Coad, 91] provide abstraction mechanisms for instance connections between classes and/or objects. The concept allows representations of situations that often occur in domains being modelled, and is an idea carried over from entity-relationship models. Coad et al state that "an instance connection is a model of problem domain mapping(s) that one object needs with other objects in order to fulfill responsibilities". The multiplicity of an instance connection is specified on the ends of the line which relates two objects graphically.

### 2.1.7 Message Connections

A message connection represents a relationship between two objects where one requests a service and another provides a service. The parameters passed may also be shown. Message passing is explicitly supported by all methods except OMT. This is a limitation as the representation of scenarios [Goldstein, 90] [Hufnagel, 92] is difficult without representation of messages. Although, message passing can be shown in HOOD and ASG, scenario representation still poses a problem as objects in these methods and tools cannot appear in multiple sheets.

### 2.1.8 Scenarios

A scenario is a sequence of events, that represents a certain behavioural aspect of a system. A scenario representation consists of a subset of objects from the entire model which participate in the particular thread of execution being represented. The messages or invocations are sequentially enumerated based on the scenario steps. A scenario representation of the behavioural aspects of a system, may be used to identify responsibilities, develop test cases, understand implementation issues, etc. The use of scenarios throughout the life cycle is discussed by Hufnagel et al [Hufnagel 92].

## 2.2 Features of Object Oriented CASE Tools

CASE (Computer Aided Software Engineering) tools can be broadly described as computerized assistance for developing or maintaining software. CASE products can be categorized along many different criteria. Among them are how well they support various life cycle phases, what method is supported, etc. In this section tool features which support the object oriented paradigm are discussed. Again, useful concepts that may not be common are also discussed.

### 2.2.1 Development Life Cycle

Each tool implicitly or explicitly supports a software development life cycle. Simply stating the fact that an object oriented methodology is being supported does not attest to a certain development life cycle. The layout, architecture and design of a tool limits and defines the life cycle of the development process. However, in most cases the tools isolate the analysis or design steps, followed by code generation, and can be individually tailored to fit a life cycle. This lack of commitment may also lead to failure in terms of being a tightly integrated tool, supporting a methodology, thoroughly.

Criticisms of the waterfall models have pointed out its inability to support the development of large, complex and evolving systems [Boehm, 81]. Several alternative models are evolving [Davis, 88]. Boehm suggested the "Spiral Model of Software Development and Enhancement" [Boehm, 1988]. The major phases of development in this model are planning, risk analysis, engineering and customer evaluation. The development process essentially spirals through each of these phases as development proceeds in an evolutionary manner. This implies that the product of each spiral of the development process may be required during the next spiral and the products of each phase of a spiral are critical to the next phase. This can also be seen as a process model where work on different components of the software development process may progress concurrently. The radius of the spiral can be considered to be the cost and/or schedule.

Most of the tools surveyed make products of the analysis phase available for design but once code is generated (if and when supported by the tool) the relationship between analysis and design documents and code is severed. The code generated is usually simply the class interface or skeleton code as can be derived from analysis and design documents, and requires much addition in order to make it workable. The modifications and additions made to the code are often not reflected in the analysis and design documents rendering them of less value for the next spiral without going through them manually to ensure consistency. Most of the tools surveyed are optimal for a phase within the waterfall model of software development, and do not lend themselves suitably to an evolutionary approach of software development.

### 2.2.2 Method and Graphical Notation

All tools surveyed aim at providing the developer with an environment to support, in varying degrees, the process of developing or maintaining software according to a predefined system development life cycle and methodology. Therefore, a particular graphical notation and an associated set of syntax and semantics is supported by each tool. In most cases the notation is explained in detail, through examples, in the texts and papers written on each methodology. Concrete formalization of the graphical notations are not available [Champeux et al, 1992] due to the heuristic nature of analysis and design, and the complexity and diversity of situations involving software development. All proponents appear to have selected the route of providing the rules, regulations and relationship to associated object oriented concepts through examples. The mapping of concepts such as aggregation. classification, etc. to a graphical representation is usually shown by the examples.

Additional textual notations to the set of representational mechanisms provided almost always positively augment the modeling process, but such augmentations are difficult to make, non-transferable through the tool to other phases of the life cycle, and often lead to inconsistencies in reference to rules checked by the tool. The previous discussion and Figure 3 discuss the abstractions.

### 2.2.3 Scope

The scope of a tool may be loosely defined as the activities and phases of development which the tool supports. All the tools surveyed cover the analysis step, and therefore, to a degree in the object oriented paradigm the design step. Some span the step of code generation too. In most cases a data dictionary is made available for the ensuing steps. Code generation is usually limited to the generation of an initial and partial skeleton of the objects interface in languages such as C++, Ada, etc. Direct mapping from the graphic notation to the language must be possible to achieve code generation [Buhr, 84].

Although the primary function of the tools surveyed was requirements analysis, none of the tools provided any support for maintaining any textual documents containing requirements and therefore no support for identifying the relationship between responsibilities or requirements of the system under development and the analysis documents. The assumption was made that the development process begins with the preparation of graphical analysis documents. This assumption can lead to serious inconsistencies in the final product.

### 2.2.4 Products

The product of analysis or design using the tools surveyed are models which consist of various graphical and textual components. A common component of the models in the tools surveyed are object diagrams. These diagrams allow the developer to identify objects in the environment or system and their features. OODLE allows the representation of only one object in each object diagram. All the tools allow multiple sheets or drawings containing objects to define the entire model. Some of the tools permit intersection between the sheets allowing an object to exist in multiple sheets. This almost always becomes necessary as the number of objects in the model proliferates. This is also necessary in order to ensure that every aspect of an analysis or design is not represented in a single diagram and also allows for multiple views. The benefits are avoidance of clutter and diagrammatic complexity as well as providing the ability to focus on representation of different characteristics separately. One common feature identified in all the tools are the services or operations of objects. Most of the tools but not all also allow for the specification of attributes (Figure 3).

Most of the tools also allow the developer to identify relationships between objects such as specialization or inheritance, aggregation or assembly, and associations or instance connections. Ada structure graphs [Buhr, 84], OODLE [Shlaer, 90], and HOOD [HOOD Working Group, 89] tools also allow the specification of data flow between objects. A variant of this approach are message connections supported only by OOATool. Textual additions are permitted in the diagrams of all tools as explanations or annotations.

Data dictionary support is provided in almost all the tools. The Teamwork tools by Cadre Technologies provide extensive data dictionaries with all their modelling tools. The data dictionary can be used to add textual explanations for each object as well as for its attributes and services. The tools supporting HOOD also provide semi-formal specification of each object. This textual component is called the object description skeleton (ODS) [HOOD Working Group, 89]. The ODS not only provides data dictionary support but also provides a means of making the transition from diagrams to code. The contents of an ODS are discussed in detail later. The ODS usually generated after drawing object diagrams is initialized with all the design data that have been put into the HOOD object diagrams, and can then be further refined by the developer.

Code generators are available in Teamwork/OOD, Teamwork/Ada and C++ Designer. The code is developed from the information available in the design documents. In case of the code generators which generate C++ code this information is usually only enough to generate a skeletal interface for the classes. Although this does provide a degree of support to a programmer, the marginal advantage is not significant. However, significant advantage may be achieved if additional code management features are added to the code generators. The Ada code generated by Teamwork/Ada is considerably more supportive to the programmer, as various items other than Ada packages and tasks are identified.

### 2.2.5 Architecture

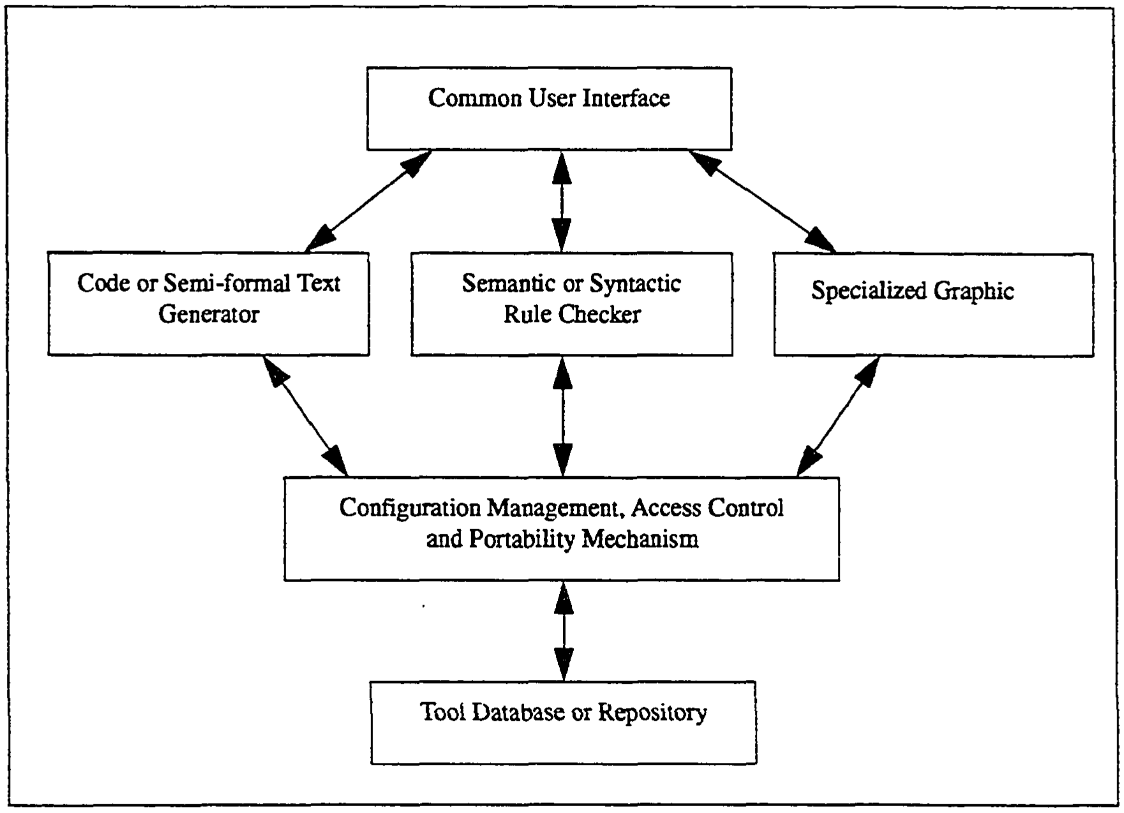

Common User Interface
Code or Semi-formal Text Generator
Semantic or Syntactic Rule Checker
Specialized Graphic
Configuration Management, Access Control and Portability Mechanism
Tool Database or Repository

**Figure 1. System Architecture of Typical CASE Tool**

All the tools surveyed appear to have a similar architecture because of the nature of the service provided. A generalized breakdown of the tools elementary components is as follows and also shown in Figure 1.

a) Underlying database or repository containing model data:

The CASE repository provides a single place for storage and access of all information pertinent to the development process. The repository usually provides services such as data dictionary, transaction control, security and multi-user support.

b) Specialized graphic editors supporting a specific set of abstractions:

The graphic editors usually provide support for each type of graphical components of a design methodology. The graphic editors are integrated with the underlying database so that various items within each graphical component is automatically stored and accessed. Interaction between the database and the editors are completely transparent to the users and cannot be modified.

c) Semantic and syntactic rule checking mechanism:

The rule checking mechanism ensures conformance to diagram rules and consistency among diagram components. The rule checkers are invoked separately after diagrams are developed.

d) Code or semi-formal text generators:

The code or semi-formal text generators generate code in languages such as Ada or C++ from the models developed, once the models are checked for consistency. The code generators simply generate the partial skeleton code in separate files which are no longer maintained by the tool as a component of the development process.

e) Configuration management, access control and portability mechanisms:

Configuration management tasks such as version control and multi-user access is controlled and provided by the different tools. The degree of support varies depending on the tool.

### 2.2.6 Re-engineering and Reverse Engineering

Reverse engineering attempts to create a representation of software source code at a higher level of abstraction [Pressman, 92]. Reverse engineering tools attempt to develop design documents from source code. These tools extract data from source code based on which data, architectural, and procedural design information is derived. Re-engineering or reclamation uses such derived information to modify and upgrade existing systems as well as to unearth glitches and operational or performance related problems.

Although, only the Teamwork tools provide a reverse engineering feature, it would not be difficult to develop tools to generate design documents of any of the modelling techniques surveyed. It is clear that modelling techniques such as OODLE [Shlaer, 90] or ASG [Buhr, 84] from which code can be generated lend themselves to be targets of reverse engineering. Mapping to design documents of such methods are easy to obtain from code. Lexical analyzers are used to identify tokens within the source code, which are graphically depicted using the modelling rules. Accordingly, Teamwork/Ada and Teamwork/OOD provide reverse engineering tools. Although the models derived by these tools are methodologically consistent and correct, the graphical objects are aesthetically not very clear and usually cluttered. This problem is however easily resolved by slight modifications and by adjusting the layout of the graphical objects manually. The advantage and assistance provided to the developer or maintainer in terms of increasing the understandability of a system and reducing the cost and effort of maintenance is considerable.

### 2.2.7 Reuse

Reuse pertains to the use of software components in multiple problem solutions. Current approaches toward reuse have shifted considerably towards a planned and proactive reuse method rather than reuse being an incidental result. The difference in approach alters the development process as reuse becomes a primary design criteria. Bollinger et al define reuse as "the process by which existing software work products (which may include not only source code, but also products such as documentation, designs, test data, tools and specifications) are carried over and used in a new development effort, preferably with minimal modification" [Bollinger, 90].

Biggerstaff et al [Biggerstaff, 87] identify four fundamental problems from the operational perspective:

a) finding components: locating similar components if not exact matches

b) understanding components: a graphical model of the component and/or specifications in order to provide the developer with a mental picture

c) modifying components: the vital issue in reuse as a component in order to be reused almost often requires modification

d) composing components: designing and developing components in a manner which explicitly addresses both the system under design and future reuse.

The method used for development and abstractions chosen for representation are critical for all the problem issues stated above. Object-oriented development provides support for reuse by concepts such as encapsulation and inheritance. Tools help by providing mechanisms for finding components, such as browse features, and for understanding components, by providing access to and mapping between products from different phases of development. Although none of the tools surveyed explicitly address the issue of reusability of models and components, the tool features for retrievability (such as browse) and maintenance of relationships among specifications defines to a great extent the degree of support the tool will provide in terms of reusability.

### 2.2.8 Navigation and Use

The ease of use will determine to a great extent whether or not a CASE tool will be adopted by developer's. Various studies have indicated that anywhere between 70 to 90 percent of CASE tools and techniques never get used in organizations even if they are acquired [Kemerer, 90]. The actual problem that often leads to CASE disuse is associated with learning the use of a tool or technique. Evaluation through the use of learning curves indicate that improved performance does not happen until much later in the learning process [Kemerer, 90].

Important factors related to ease of use are the graphical user interface, traversal or navigation between the related components, standardized terminology, etc. Method related issues regarding ease of use are abstractions that make common sense, are not too complicated, are not counter intuitive, and clearly stated guidelines.

All the tools surveyed except OpenSELECT run on graphical user interface platforms. The Teamwork set of tools all have the same initial interface and a highly standardized look and feel. Mastering the use of one tool makes the use of the other tools considerably easier. Each tool also supports methods that have been well defined and follow the authors terminology strictly. The tools also adopt standard techniques of access and use. However, certain aspects of use of all the tools are at times clumsy and counter intuitive and more attention needs to be given in terms of ease of use during development of the tools.

### 2.2.9 Consistency Checking

Checking the models and components for semantic and syntactic integrity and correctness are a vital support provided by all the tools. This assists development efforts considerably both in terms of size and complexity. Part of the consistency in the models is imposed throughout development by the limitations of the tool. A tool limits the different forms of graphical and textual components that are available to the developer, thus forcing him to stay within the boundaries of a notation.

However, proper use of the graphical and textual components are another factor altogether. The rule checkers provided by different tools are designed to ensure that modelling rules are obeyed. The checking mechanisms usually produce both errors and warnings. Some checkers can even provide support for design heuristics for improving productivity and quality usually in the form of warnings. In order to be effective the rules checked by a rule checking mechanism must be clearly enumerated or listed and the developer must have a clear idea regarding these rules.

### 2.2.10 Version and Configuration Management

The umbrella activities which span the entire development process and are aimed at managing the software development process as well as the components, tools, products, etc. of each phase are configuration management activities [Pressman, 92]. The items that fall within the scope of configuration management can be called software configuration items [Bersoff, 84].

Version management is aimed at managing different versions of configuration objects that are created during the development process. This allows the developer to have several alternatives at different stages of development and allows the developer to clearly identify objectives at different stages of development [Pressman, 92].

The tools surveyed have a common basic approach to configuration and version management. All software configuration items within a scope of a development effort are contained in a model. The components of a model can be graphical diagrams or textual specifications which are interrelated, can be baselined or copied in whole. Each model may have several copies or specific versions. HOOD tools allow hierarchical combination of two or more models. Other tools allow copying some of the components over to another model thus allowing combinations. Combining a set of objects developed in a model with another model is also possible in some tools.

### 2.2.11 Portability

Portability pertains to the aspect of transporting models within a platform and between platforms. The portability of models developed on the tools in the DOS and MS Windows environment is relatively convenient as the model files and in some cases associated database files can simply be copied over to another location where the tools can be used to successfully access these files. OOATool files provide an additional useful feature as the model files can be transported into an Unix environment in the same manner as the model files are in ASCII format. However the Teamwork set of tools require the conversion of the models into CDIF(-CASE Data Interchange Format) format in order to be transported.

## 2.3 Tools

This section provides a discussion specific to each tool in the survey. The discussion focuses on individual attributes of each tool and attempts to present the tools features and design.

### 2.3.1 OOATool

The tool allows the developer to develop graphically a model of the system being defined. A model is seen as a collection of drawings. The union of objects, structures, associations and messages in all the drawings forms the model. Alternatively, each drawing can be seen as a selection of objects and associated material from the entire model, all other objects and related material being temporarily hidden. This is a useful concept provided by the tool, as it allows the developer to work from the middle up or down, and does not limit him to top-down or bottom-up development. This is also the main mechanism for providing view abstraction and can be used for developing scenarios. The tool also provides an editor for specifically changing the specifications of an object.

OOATool provides a filtering feature that allows the developer to limit the contents of the object which will be seen in each drawing. This provides further view abstraction. However, filters apply only to objects, attributes and services. Instances in OOATool are related to attributes whereas messages are related to services. Therefore, view abstraction applies to instance connections and message connections. This can get cumbersome and may be viewed as a limitation, which could be easily alleviated by providing filters for everything.

Coad et al [Coad, 91] also suggest another mechanism for view abstraction, supported by the tool. Several drawings involving the same objects can be created providing a layered view of the objects. The layers suggested are the object layer consisting simply of objects, the subject layer consisting of the objects clustered as subjects, the attribute layer which adds the attributes and instance connections and the service layer which shows the services and the messages.

Hierarchical abstraction is also possible to an extent. The feature used to incorporate hierarchical abstraction is called a subject. A subject is a group of objects or subjects, which can essentially be collapsed into one component. These can be seen as a loose interpretation of assembly structures. However, this concept is not directly used for traversal between drawings as is the case for HOOD [HOOD Working Group, 89] tools or tools supporting functional methodologies.

Large domains represented by many objects and interactions can be difficult to represent in the OOATool. Although, a utility is provided to be able to navigate within large drawings which provides an overview of the drawing, management of objects in such a drawing are a definite drawback of the tool. Using the object layer to view such drawings provides a simplistic solution of this problem, but only to a limited degree.

Consistency to a model is automatically provided by restricting the user to a limited set of options. This is implicit. However, further consistency checking mechanism is provided which allows the user to critique models or drawing.

**2.3.2 C++ Designer (Select)**

The C++ Designer supports the method of object oriented software development proposed by Rumbaugh et al [Rumbaugh, 91]. However it does not support the entire method which consists of the object model, functional model and dynamic model. It supports only the object model development and partial code generation from object models. This tool has a similar construction to the OOATool. Models in OOATool are called projects here. A project consists of multiple files similar to drawings in OOATool. The union of all the objects and related material in all the files defines the objects in the project. However, this is implicit and the view abstraction is not achieved simply by hiding objects and other items. All objects, structures, associations, links, etc., are entered into a dictionary of the project. Any item may be reused from this dictionary in any drawing by browsing through the dictionary. Editors specific to objects, associations and structures are available.

Filtering features are also available in the C++ Designer. Attributes, operations and super-classes can be selectively displayed. Although this allows a lot of objects to be shown in a diagram, it does not sufficiently alleviate the problem of managing many objects and interactions in a domain. The C++ Designer also provides an additional feature of being able to zoom, very similar to that used in cameras. This feature increases the manageability of projects containing numerous objects and interrelations. However, object information will hardly be readable if a file consists of too many objects.

Traversal of diagrams within a project is possible through the use of the browse feature of the dictionary. A usage list is provided for every object indicating the objects existence in various files. The user can easily navigate between all the files containing the object. Traversal is not directly supported along aggregation or classification structures, but the usage feature may be used to achieve this effect. Hierarchical abstraction is not supported by any other way in this tool.

Associations and links between objects are useful to show relationships between objects similar to instance connections in OOATool. Behavioural abstraction of different views of the system are not explicitly supported by message connections by Rumbaugh et al [Rumbaugh, 91] neither in their method nor in any of the tools. Hence, scenario representation using this tool is not possible, unless lengthy textual attachments are made to each file.

### 2.3.3 Teamwork/Ada

Teamwork/Ada supports the object oriented design method for developing systems in Ada proposed by R.J.A Buhr [Buhr, 84]. This notation provides a one-to-one mapping between a set of graphical elements and the corresponding features of the language Ada. The graphical component of this method are called Ada structure graphs (ASG) and the tool provides an editor that supports this method completely and allows the developer to create and maintain ASGs. The graphical elements of this method are packages, subprograms, tasks, clouds, data objects, etc. A powerful Ada code generator and a reverse engineering tool are also provided with Teamwork/Ada.

A Teamwork/Ada model consists of a set of ASGs which describe each component of a system being developed and specifications showing how these components are interrelated and can be accessed. A model is developed in a hierarchical manner where the top level view represents the main library units and their interdependencies. The next levels specify each component in further detail. The top level diagram is called the context ASG. Text can be added to enter specific source code or logic, exception handlers and declarations within a component which are carried over to the code generation phase. Both hypertext based traversal within the model and direct access to a single ASG though the use of a menu are supported.

All version control, baselining and configuration management support is identical in all the Teamwork tools. Combining different models is cumbersome and difficult in the tools. The consistency checking mechanism which is provided, supports an elaborate list of rules specified by the method. Although, this is necessary in order to be able to generate code and sometimes useful, analysis tasks are made difficult, due to the imposed rigidity. This also causes the developer to take a programming in the small approach from the very beginning. Hence, the tool and the method lose effectiveness for most of the software development process other than as an automated coding tool. It is possible that an experienced developers productivity will hardly be enhanced by this tool. However the reverse engineering component makes it extremely effective for maintenance and enhancement purposes.

### 2.3.4 Teamwork/HOOD and OpenSELECT

The use of Teamwork/HOOD and OpenSELECT are identical in many aspects since they both support the same method. Hierarchical object oriented design or HOOD is specified in detail in the HOOD User's Manual developed and published by the European Space Agency [HOOD Working Group, 89]. However, configuration management, access, traversal, and the user interface for the tools are quite different. The Teamwork tool has the look and feel common to the Teamwork tools while the OpenSELECT provides a simplistic windowed environment (not MS Windows).

A Teamwork/HOOD model or OpenSELECT model consists of a collection of HOOD structure graphs and object description skeletons (ODS) derived in part from the HSGs {HOOD Working Group, 89]. A graphic editor is provided for the HSGs and a text editor is provided for the ODSs. The elements of an HSG are objects, uncle objects, operations, arrows, exceptions and data flows. A HOOD object may be of the types 1) passive, 2) active, 3) environment, 4) virtual and 5) operation control. Active objects contain at least one constrained operation. The constraints on operations may be of different types based on the execution request: 1) functional activation, 2) highly synchronous, 3) loosely synchronous, 4) asynchronous and 5) timed-out. The HSG does not support the specification of attributes of an object. A partial ODS can initially be generated by ODS generators in both tools, which can then be augmented. The ODS is divided into six main sections: 1) a description section, 2) provided interface, 3) required interface, 4) internals, 5) object control structure and 6) operational control structure.

The decomposition of a HOOD model [HOOD Working Group, 89] is hierarchical where the top level is called the system to be designed. This diagram may contain several objects which can be specified further in the next level. The decomposition can be seen as a tree structure based on the aggregation relationship. Each object and hence each diagram has an object description skeleton. Hypertext based traversal between diagrams and to their ODSs by the object hierarchy is supported in both tools. Direct access to any component within a model through the use of the menu is also allowed.

The hierarchical property of HOOD models [HOOD Working Group, 89] make it simple to decompose the tasks of a development effort into separate units and to integrate them later. The interface to an entire model is specified by the services or operations available at the top level or system HSG. Both tools exploit this property by providing methods of combining multiple models. This is particularly convenient in case of the OpenSELECT tool. Uncle objects and environment objects in HOOD also provide a system of tying together different models or of violating the strict tree structure when necessary.

Rule checking mechanisms for the graphical component and the ODS exist in both the tools. It is important that the graphical component be developed correctly before the ODS can be generated. The generated ODS complies with HOOD standards developed by ESA. The description of the ODS and augmentations such as pseudocode or code is in Ada syntax when possible or Ada based pseudocode. This facilitates translation of a HOOD model to Ada code. However, none of the HOOD tools provide an Ada code generator or reverse engineering tool.

### 2.3.5 Teamwork/OOD

Teamwork/OOD supports Project Technology's Object Oriented Design Language (OODLE) notation developed by Sally Shlaer and Stephen J. Mellor [Shlaer, 90]. This tool uses existing functionality in Teamwork/Ada by making minor configuration adjustments to support OODLE. The method consists of four major graphical components: 1) class diagrams, 2) class structure charts, 3) dependency diagram and 4) inheritance diagram. The graphical elements of the components are classes, logical components, functions, dataflows, etc.

The Teamwork Ada editor is modified and reconfigured to develop each of the graphical components. Traversal between the components is not well supported and expected to be achieved by the combined use of a strict naming convention for diagrams and the menu. However, traversal options between diagrams will operate properly if the naming convention is followed.

Valid C++ code can be provided as textual additions to the class diagrams in the model. The editor supports attachment of code to different graphical components and the tool ensures that the code is arranged in the appropriate order and location. The tool also provides a C++ code generator which uses information from all the graphical components and code additions as notes to create a C++ code skeleton or frame.

Rule checking mechanisms for all four types of diagrams are available. The rules supported are specified in the OODLE notation and based on object oriented programming languages, especially C++. Code may not be generated until a model is checked to be completely consistent. All Teamwork tools can also be used in conjunction with tools such as Teamwork/DocGen, Teamwork/DPI, etc. to generate reports for cross reference and various other design support documentation.

## 2.4 Conclusion of Comparative Analysis

The tools surveyed are a positive addition to a developers toolkit, because they automate processes and activities which are difficult, tedious and time consuming. They provide support for formal techniques of analysis and design, and provide data dictionary support. All tools provide some checking mechanisms. Some tools also provide skeleton code generators and reverse engineering tools. Primarily, they alleviate the problems related to management of process complexity.

However, most tools only provide a limited degree of integrated support and they do not support the entire development life cycle. Although, object oriented methods are supported, the tools are better suited to the waterfall life cycle model. They also inherit limitations of the development method used. All the tools surveyed had weaknesses in terms of handling very large development efforts, effective multi-user support and integration of distributed development efforts. Traceability suffers as a project develops and development efforts tend to lose focus on responsibilities and requirements of a system under development, when using any of the tools. Reuse of all software configuration items is also not well supported.

# 3 NOTATION FOR THE S. E. P

The goal of this research is to propose a specification for an integrated tool which will robustly support the scenario-based engineering process and object oriented concepts. A notation needs to be selected in order to specify the tool in detail. Hufnagel et al [Hufnagel, 92] and Haddock [Haddock, 93] propose a life cycle for software development using the scenario-based engineering process. An effective notation is selected or developed in this section to support different phases of the SEP life cycle. The notation also allows representation of the different aspects of a system or domain, such as the structural, behavioural, dynamic and functional aspects. The notation consists of graphical and textual components used in various phases of development.

Components of the notation are selected from well defined modeling techniques, based on representational needs of the scenario-based engineering process. The SEP notation which is presented in this section is language independent and object oriented. However, the responsibility driven and scenario based engineering process may be applied using several alternative notations. Scenarios are an essential part of this method and its associated tool, as they assist developers in analysis by identifying major roles and interactions of objects in a domain and in design by identifying responsibilities of a system under design. Scenarios also provide the developer with validation suites for testing, verification and validation.

## 3.1 Goals and Objectives

The following discussion will develop the notation to support the scenario-based engineering process of software development. The primary purpose of the notation is to provide the representational techniques, which are needed to analyze and specify a system using the scenario-based engineering process. The notation also attempts to achieve completeness in terms of representation.

- A notation or formalism must be suitable for each phase of software development.
- The notation must exploit the benefits derived from the scenario-based engineering process and be able to support its vital concepts.
- Each notational component must be able to support important abstractions necessary to achieve the representational goals of the notation.
- The notation must represent fundamental object oriented concepts such as encapsulation, aggregation, inheritance, polymorphism, etc.
- Whenever necessary, the notation should properly exploit well defined and useful concepts from any existing paradigm or technique such as structured analysis and design, entity-relationship data modelling, etc.
- The notation must successfully include and assimilate concepts identified from the survey of methods and tools which were seen to be useful, but only available in one method or more but not all.
- The relationship between each component must be clearly identified and whenever a clear, simple and well defined mapping between components or elements of components is possible, the relationship must be maintained by the tool.
- It must be possible at some point to make language specific interpretations or mappings from the notation.

## 3.2 The Scenario-based Engineering Process

The scenario-based engineering process was developed as a seamless object oriented approach for developing software. Initially, scenarios are used as a medium of behavioural abstraction to understand the domain and define responsibilities. The scenarios then become the unifying and direction providing basis for the entire development process. Scenarios are mapped or refined into tasks, as a means of transforming domain knowledge and requirements analysis into one unified set of responsibilities or tasks. Scenarios further get implemented as technological components where the initial scenarios play the role of validation suites. Harbison et al [Harbison, 93], Hufnagel et al [Hufnagel, 92] [Hufnagel, 93] and Haddock [Haddock, 93] develop and discuss this process in detail.

The scenario-based engineering process takes a three layer approach to the task of analysis [Haddock, 93], the layers being the environment layer, the domain layer and the technology layer. Such an approach allows the developer to first understand the domain, then extract the essential items and finally to focus on implementation.

The object oriented and scenario-based life cycle attempts to maintain traceability and conceptual integrity throughout a development process by adhering to the responsibilities of the system being developed. It also ensures multiple views of a system and enhances communication between different phases and developers. The process also allows the developer to take multiple approaches at development. The development may begin with the creation of new components or with the reuse of predefined components and when creating new components future reusability plays a major role. The development process may be top-down, bottom-up, or middle-out.

## 3.3 Components of the Notation

The abstractions required to specify a system completely usually combine representation techniques for the structural, behavioural and functional aspects of the system. Rumbaugh et al [Rumbaugh, 91] use object diagrams in their object model, state-transition diagrams in their dynamic model and data flow diagrams in their functional model to achieve complete representation. Shlaer et al [Shlaer, 90] use a combination of class diagrams, dependency diagrams and inheritance diagrams for the structural aspects and class structure charts for the functional aspects of a system. Coad et al [Coad, 91] use object diagrams to combine structural and behavioural aspects of a system to an extent.

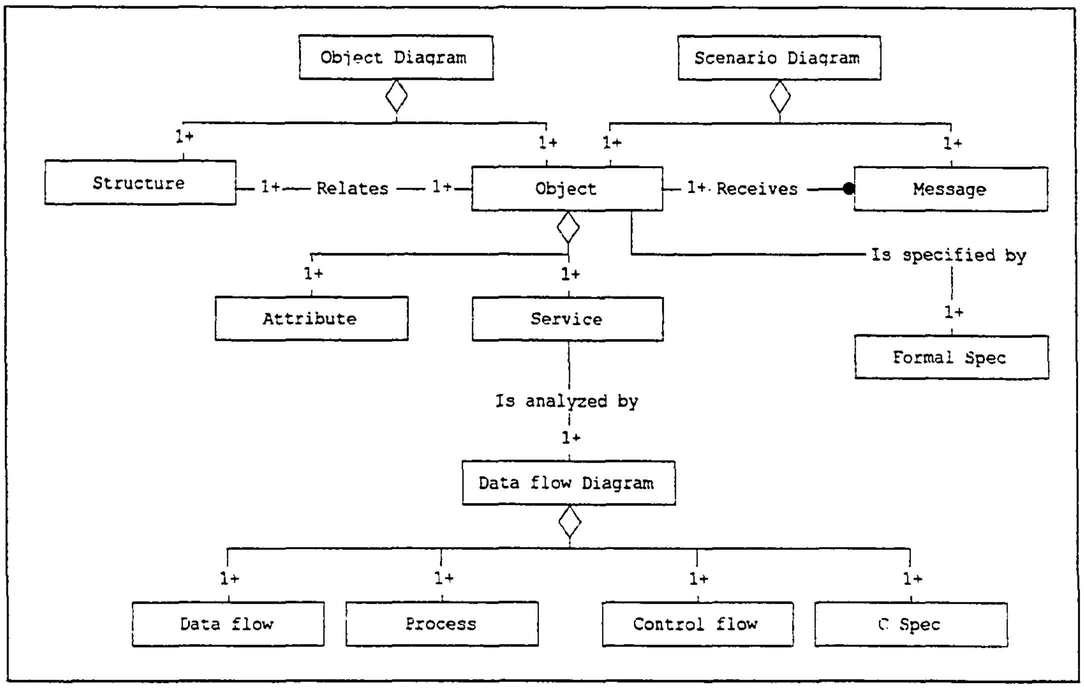


**Figure 2. The relationship between components of the notation**

1+ indicates a multiplicity of one or more, no number indicates a multiplicity of 1, and a circle indicates 0 or more. The diamond shapes on a relation indicates aggregation or the *is made up of* relation.

The following diagrams or methods of graphical abstractions have been developed or selected from the literature on object oriented and functional development of software to successfully and completely support analysis and design using the scenario-based engineering method. Figure 2. shows the relationship between the graphic representations within a model. The primary components of the notation are 1) object diagrams, 2) scenario diagrams, and 3) data flow diagrams.

### 3.3.1 Object Diagrams

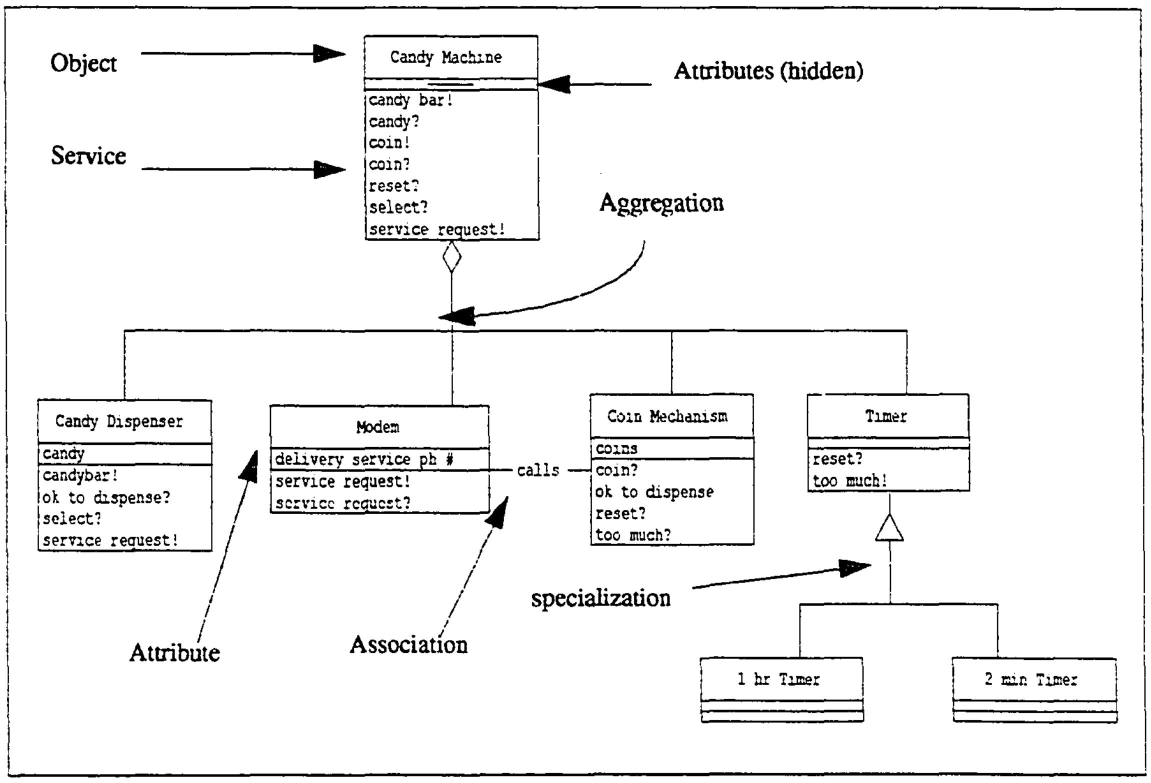


**Figure 3. An Object diagram using Object Modelling Technique (Rumbaugh) notation.**

The diagram above shows the various required notational elements for an object diagram. The different elements shown are objects, attributes, services, specialization, aggregation and association. Notation and strategies for object-oriented analysis have been developed by several authors such as Rumbaugh et al, Coad et al, etc.

An object diagram depicts the set of objects within a structure such as an aggregation, specialization or association. The notation for this purpose has been well defined in the notations of Coad and Yourdon [Coad, 91], Rumbaugh et al [Rumbaugh, 91], etc.

### 3.3.1.1 Purpose

An object diagram provides a representation for static structures within a domain or system. Objects in the system, classes, relationships between them, and characteristics of each object such as attributes and operations are shown in these diagrams. Object diagrams provide a formal graphic representation of the interface of each object and its external characteristics rather than internal details.

The representation of static structures with the use of object diagrams allows the developer to understand the domain and to communicate his perceptions with the domain expert or user. Simplicity and abstraction of the essential components and characteristics is the key principle involved. Object diagrams must be concise, easy to understand and represent a single view instead of attempting to combine many aspects of a domain or system. If the same object belongs to multiple structures then each structure should be shown on different object diagrams. Object diagrams also serve as basic design and implementation documents as well as a framework for reuse, access or integration. Design objects are to be implemented as abstract data types.

### 3.3.1.2 Symbology and Logical Elements

Figure 3. shows an object diagram where all the graphical elements of an object diagram are shown, using the Object Modelling Technique and the C++ Designer. The following elements are necessary to develop an object diagram. 1) Objects are shown by a rectangular box. The top part of the object box identifies the object and must be unique in the entire model. 2) The second part shows attributes. During analysis an attribute simply needs to be identified. During design its data type must be resolved, which can be the elementary data types of a language or other objects. 3) The third part shows services or operations which follow the same rule as far as identification of data types. 4) Structures can be aggregations, classifications or associations between objects. Lines can connect the different objects in a structure and different graphic tokens can distinguish between the three types of structures. Multiplicity of an association can be marked on two sides of an associating line.

#### 3.3.1.3 Extensions

Whether an object is active or passive may be indicated by an A or P in the left hand corner of the object identification section [HOOD Working Group, 89]. When an active object is invoked control is not necessarily transferred to the operation instantaneously. As developed in HOOD, it is required that an active object contain at least one constrained service. The constraints should define if the operations will execute in synchronous or asynchronous relation to the caller. The type of constraint may be stated beside an operation or symbols may be used for this purpose.

During analysis the data types of attributes and operations do not need to be identified. However, during design or before implementation this task must be completed. In this manner analysis specific object interfaces may be moved closer to implementation.

### 3.3.2 Scenario Diagrams

A scenario diagram depicts the data flows or message flows between objects in a sequence of events, which represents a certain aspect or behaviour of a system. A message carries the parameters for the invocation of a specific service. The sequence of invocation of services or operations of objects to achieve representation of a behaviour or task is the basic approach for software development in the scenario-based engineering process.

### 3.3.2.1 Purpose

The scenario diagrams are used to show the behavioural aspects of the system or domain. Many possible scenarios may be developed from a set of given objects. However, not all scenarios are useful or meaningful. Scenarios allow the developer to separate and understand each important aspect or function of the system. At the same time a scenario may also identify erroneous requirements or incorrect analysis objects. Identification of objects is usually a heuristic process and clear definitions of a correct or incorrect set of objects is often impossible. But, if it is impossible to achieve a given requirement scenario using the objects in the object diagrams and their services, then the developer will be able to assess whether the objects and structures do not fit the purpose or the services are not sufficiently developed.

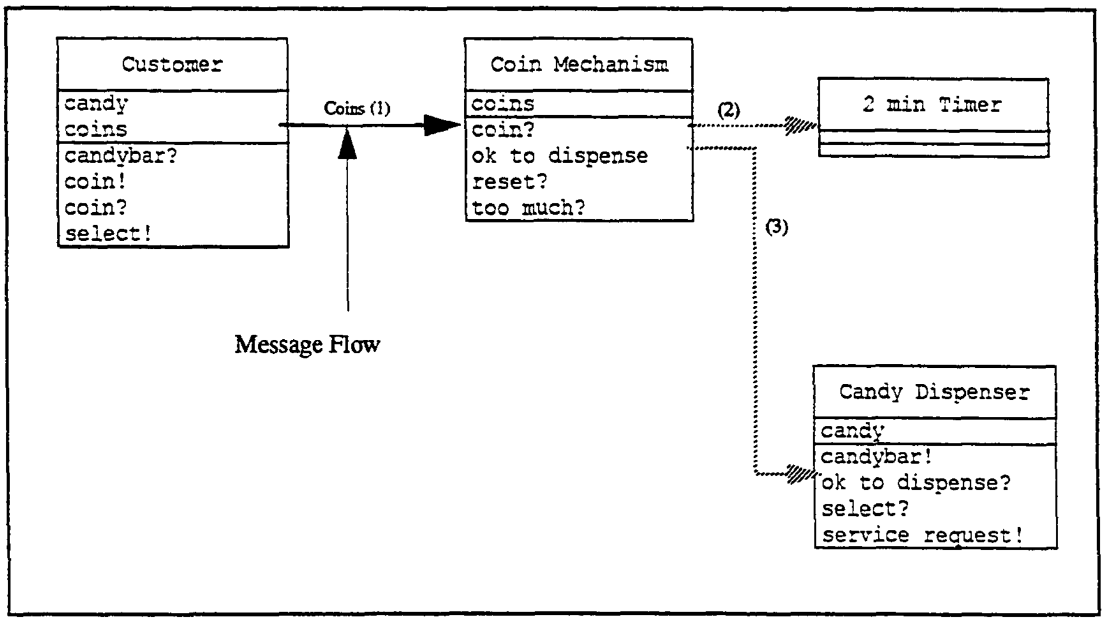


**Figure 4. Possible representation of a scenario**

Messages are shown by arrows. Solid arrows are data flows while dashed arrows are control flows. Message arrows point at the service being invoked.

Analysis scenarios can be used throughout the life cycle and can be refined into separate tasks and mapped into capabilities. This provides a measure to the developer in terms of project completeness and progress. Haddock [Haddock, 93] points out that scenarios can exist in three layers when analyzing the system. The layers specified are the environment layer, the domain layer and the technology layer. Scenarios in all the layers, especially the first two layers can be developed using the same notation. The difference in the scenarios in the three layers are in terms of the objects which participate in the scenarios. In the environment layer the objects may belong to the system or the external domain, and their definition need not be too detailed. In the domain layer the objects participating are primarily within the system although an environment object may be an initiator or terminator. In the technology layer the objects must come from a code repository.

Scenarios are also useful in later phases of the life cycle. Scenarios are excellent specifications of testing guidelines. Each scenario may be treated as a validation suite. Scenarios can also provide a framework for integration activities.

#### 3.3.2.2 Symbology and Logical Elements

Figure 4. shows an example of a scenario. A scenario uses objects from one or more object diagrams in the model and an object may participate in multiple scenarios. A sequence of invocations are shown. This is achieved with the use of message arrows connecting different objects. The arrow leaving an object implies that a service was received from that object. The arrow points to a service in another object. This service is being invoked. A directed loop pointing to a service within the same object is permitted in order to represent a sequence of services being invoked in the same object. A broken or dashed arrow represents a control flow message. The parameters are shown on the arrow and may have been derived from the services received before the one being invoked. The sequence of operations are shown by enumerating the arrows in order. Therefore, an arrow is labeled by the couple (n, parameter list) where the n-th service in the scenario is being invoked. The parameter list may be composed of data or control flow.

### 3.3.3 Data Flow Diagrams

A data flow diagram depicts the function oriented view of a system and was used primarily for analysis using structured or functional methods. The notation and its use has been explained in detail by Yourdon [Yourdon, 89]. Data flow diagrams are used in this notation primarily with respect to services or operations. The use of data flow diagrams in object oriented analysis is a concept taken from the functional model of the Object Modelling Technique. Some data flow diagrams are shown in figure 5.

#### 3.3.3.1 Purpose

The primary purpose of data flow diagrams in this notation is to provide a means of analyzing the functional aspect of each service or operation of an object. A data flow diagram is a modelling tool used extensively by practitioners for hierarchical and functional analysis of a system. Although proven ineffective and not sufficient for large and complex systems, data flow diagrams are excellent tools for analyzing small, relatively simple and well defined domains. The operations in an object model or a scenario model are treated as black boxes which have specific inputs, either external or internal. When invoked the operations take a certain set of actions and possibly return a value. A data flow diagram is an excellent abstraction tool for analysis of each such service or operation as a message flow can be interpreted as a data flow.

Although, practitioners in the object oriented paradigm usually stay away from the use of data flow techniques, it is undeniable that there is a functional aspect to any system, and the use of functional modelling techniques is most appropriate for understanding this aspect. As the object model will essentially reduce the entire system to numerous functional components, the developer can use data flow diagrams to analyze each component instead of treating the entire system as a single functional entity, thus effectively bypassing the problems associated with functional modelling techniques.

### 3.3.3.2 Symbology and Logical Elements

Data flow diagrams are directed graphs where a vertex may be a process, a data store or an external entity. A process is represented by a circle, an external entity by a rectangle and a data store by a pair of parallel lines. Each arc in a data flow diagram represents a data flow. A data flow may be singular or composite. Each process may be decomposed into more data flow diagrams. In this way a strict hierarchy is imposed on the system being decomposed.

The purpose of a process or transform is simply to produce outputs based on inputs. Processes are named according to their functions and a numbering scheme is usually followed which reflects the hierarchy. Each leaf process may be further specified in a process spec (specification) containing pseudocode or code. The top level process or the context data flow diagram contains a single process representing the operation or service being analyzed. Therefore, the context data flow diagram of a service shows the actors or external objects a service, data stores in the environment or environment objects and the data flowing between them and the operation or service. These data flows must be consistent with the data flows in the scenario diagrams. Consistency is required between the flows related to a parent process, and those in a child data flow diagram. Composite flows may be decomposed as multiple flows representing each component flow in a child data flow diagram.

The external entities may be actors or active objects which initiate the service or receive the values returned. The data stores may be attributes of the object to which the service belongs. A data flow may not exist between data stores as a data store is not a functional process. The actors or data stores are named in the data flow diagram using a convention such as object.service or scenario.message arrow.

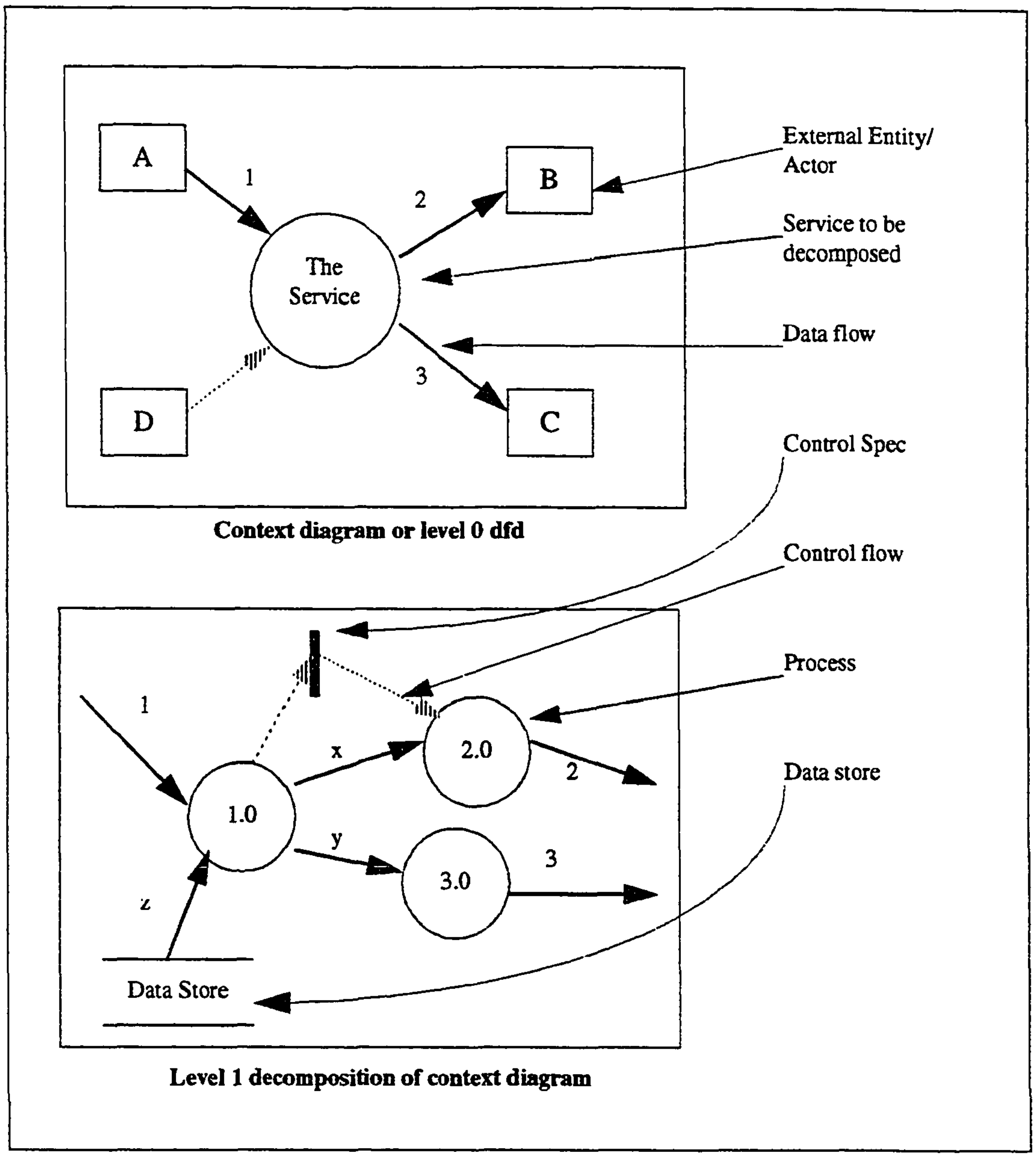


**Figure 5. Data flow diagram for each service**

Functional decomposition of the services provide an analysis method for detailed analysis of each service. Notation and strategies for structured systems analysis using data flow diagrams have been developed in detail by E. Yourdon in his book "Modern Structured Analysis".

#### 3.3.3.3 Extensions

Data flow diagrams have been extended for use in the modeling of real-time systems by Hatley and Pirbhai [Hatley, 87] into a modeling notation called control flow diagrams. These extensions may be used for analysis of services which are constrained in active objects. The extensions in terms of notation or graphical elements consists of control flows shown by arrows composed of broken lines in order to distinguish them from data flows and control specs (specifications) which are shown by vertical bars. Control flows are discrete values that may take on one of a finite number of known values. A control flow from a process must flow into a control spec bar and not to another process.

A control flow emanates from a process when a data condition occurs, thus crossing over into the control structure of a control flow diagram. The control spec contains a finite state machine in the form of a process activation table or a decision table with the use of which a process control output is generated to activate or deactivate a process. A control flow or a set of control flows may be interpreted as an event while a process control output may be seen as an action.

## 3.4 Additional Components or Alternatives

Although the three modelling techniques specified in the previous section are sufficient to analyze and design a domain or a system using the scenario-based engineering process, several additions or augmentations to the notation are possible. Structure charts may be used for further specification and design of services using a functional approach. State charts, state-transition diagrams or any variation of finite state machines may be used for behavioural specifications. Petri Nets may prove useful for behavioural specification of concurrence or parallelism and certain other complex situations.

### 3.4.1 Structure Charts

A structure chart shows modular decomposition of code into functions and procedures. Other than depicting the hierarchical organization of modules, a structure chart also shows parameters passed and values returned [Page-Jones, 88]. Structure charts are used in this process to convert analysis information of each service or operation to design documents. The use of structure charts is optional and a developer may find it easier to go to coding when data flow diagrams of services are complete. Most of the time structure charts may be entirely unnecessary as the implementation of the service may not be complicated, but if the data flow diagrams in the analysis of a service exceed more than two levels of decomposition a structure chart may be useful.

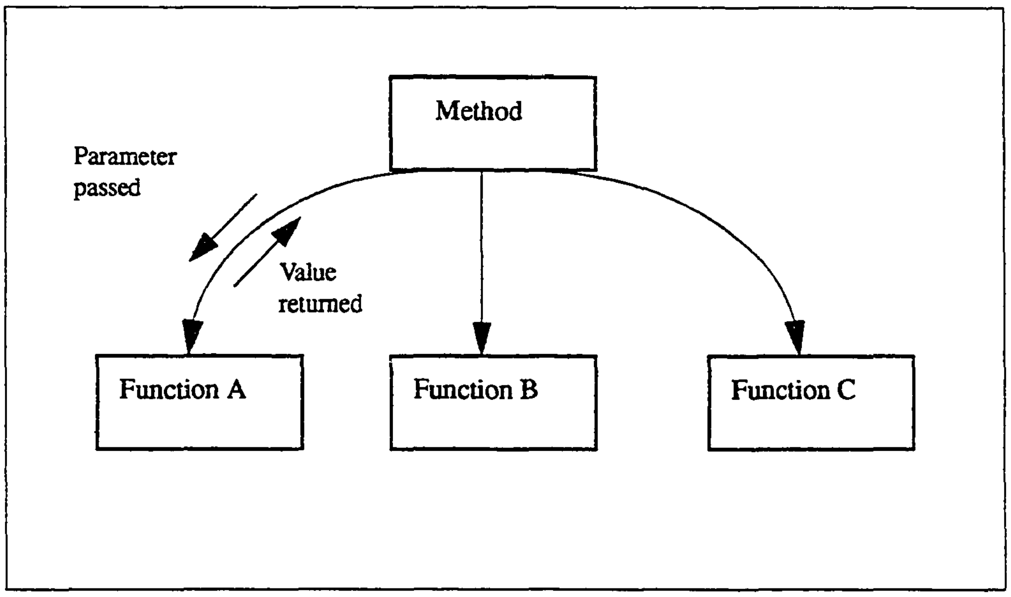


**Figure 6. Hierarchical Structure Chart of Method analyzed by Dfd**

Structure chart showing the modular decomposition for the desired implementation of a method. Notation and strategies for structure charts have been clearly and completely developed by M. Page-Jones in his book "The Practical Guide to Structured Systems Design".

A hybrid version of structure charts have been used in class structure charts in OODLE. The notation selected for SEP uses structure charts as explained by Page-Jones [Page-Jones, 88], instead of attempting to develop new notation in order to avoid unnecessary innovation.

### 3.4.1.1 Purpose

The role played by structure charts in this notation is optional and applies only to the design of services or operations. A structure chart is used to depict the modular structure of a service, and an important tool for design using a functional approach. The partitioning of a service into functions or modules and their hierarchical organization is shown using a structure chart. The parameters passed between modules and the values returned are also shown. Development of these diagrams may allow code generators to be developed which produce code which is far more complete than that developed by most of the tools surveyed. A structure chart is shown in figure 6.

### 3.4.1.2 Symbology and Logical Elements

The structure chart is intuitive and simple. The main graphic elements of structure charts are modules represented as rectangular boxes. They are arranged on a page in a hierarchical organization, the service or operation itself being the top module. If a service is used from another object (such as a friend function) then the function is named as object.service. If a function is used whose use remains restricted to the object than it is simply named meaningfully. Arrows connect different modules going from the calling module to the called module. One or more data flow symbols showing the direction of a flow and the name of the data flow are attached to the arrows connecting the modules. The key concept in developing structure charts is to identify or map modules from processes in the data flow diagrams with cohesion and decoupling being the guiding principles.

### 3.4.2 Finite State Machines, State Charts

The use of state transition diagrams or finite state machines for understanding the behavioural aspects of an object has been described in detail by Rumbaugh et al [Rumbaugh, 91]. The theory is discussed in various texts and papers including one by Sudkamp [Sudkamp, 88]. Rumbaugh et al augment traditional state transition diagrams to be able to represent a complex system concisely. This is done by the use of nested state diagrams, where each state can be expanded as a lower level state transition diagram.

A state defines the status of an object in an interval between two events, where an event may be defined as a stimulus or input to an object. The state of an object changes based on an event and the current state of the object. This is called a transition. Rumbaugh et al introduce further details in a state diagram for conditions, operations, state aggregations and state generalizations.

### 3.4.3 Petri Nets

Petri nets are a graphical and mathematical modelling tool which can be used for describing various kinds of systems. Petri nets allow representation of systems that are characterized as concurrent, asynchronous, distributed, parallel, nondeterministic, and/or stochastic [Murata, 89]. The use of tokens in Petri nets allow representation of dynamic and concurrent activities in a system and can, therefore, be used to study various behavioural aspects of a system or object. Petri nets can also be used in place of state-transition diagram because they are also capable of representing situations where state-transition diagrams have been found to be insufficient.

Murata [Murata, 89] describes Petri nets as "a particular kind of directed graph, together with an initial marking. The underlying graph of N of a Petri net is a directed, weighted, bipartite graph consisting of two kinds of nodes, called places and transitions, where arcs are either from a place to a transition or from a transition to a place." Places are graphically represented as circles and transitions as bars or boxes. The theory and various applications are also discussed in detail by Murata [Murata, 89].

### 3.4.4 Formal Specification

A formal specification language such as Z [Spivey, 92] or VDM may be used to add semantic information to the object diagrams. Giovanni et al [Giovanni, 92] describe a method of integrating HOOD by means of Z formal specifications. A similar approach may be developed for including formal specifications to the scenario-based engineering process.

The integration of formal specifications to an object oriented development process has several benefits. The abstraction level at which the formal specifications are applied is already identified by object oriented analysis or design. Each object in an object model have well defined functionalities and they are relatively small units within a system. Specifications can be developed for one object at a time. Therefore, each object will possibly need to be specified by a set of formal schemas. Formal schemas also provide a behavioural specification of an object and a guideline for implementation, testing and maintenance. In the case of reuse libraries formal specifications can be used instead of code to understand the actual behaviour of an object.

# 4 A CASE ENVIRONMENT TO SUPPORT THE S. E. P

A Computer Aided Software Engineering (CASE) environment or tool for development of software systems using the scenario-based engineering process is described and discussed in this section. The scenario-based engineering process is an object oriented and responsibility driven software development paradigm [Haddock, 93] [Hufnagel, 93]. Scenarios, as described earlier play a primary role in the entire development life cycle. The life cycle of the scenario-based engineering process is discussed in this section, in order to illustrate the use of the tool.

Dart et al [Dart, 87] classified software development environments into four main categories: 1) language centered environments, 2) structure-oriented environments, 3) toolkit environments and 4) method-based environments. Language-centered environments are built around a language and structure-oriented environments allow direct manipulation of structures. Both of these categories of tools are support for programming-in-the-small tasks. Conversely, toolkit environments provide a set of tools supporting programming-in-the-large activities only. Method-based environments such as the ones surveyed in chapter 2 focus on support for a range of activities, primarily analysis and design, based on a specific method or paradigm of software development. The tool for the scenario-based engineering process attempts to cross the boundaries of this classification and to provide broad ranging support, from programming-in-the-large tasks to programming-in-the-small tasks.

Each task or activity in the software development process can be independently supported with the use of individual CASE tools. However, maximum benefits can only be realized if the tools are completely integrated and the information interchange between tools is fluid. Although, much research and progress is being made in the direction of integration and various frameworks have been proposed, full integration has hardly been achieved. The primary problem in this regard is the lack of standardization without which integration is impossible. A straightforward solution to this problem can be the provision of an entire array of tools developed with integration in mind; in simpler terms a tool set containing all the necessary individual tools with defined communication links. In order to achieve this, notations for support of abstractions in each phase of the development process as well as the relationship between each development component must be clearly specified. A shortcoming of such an approach is the lack of alterability which can be somewhat alleviated by diminishing the need for too much modification and by making the provision for certain adjustments and tuning.

The underlying goal of this tool is to provide access to technology and information, and to serve as catalysts in developing creative and productive capabilities in software developers. The tools technical approach is to identify important system development principles and associated evaluation criteria. The tool's management approach is to identify human needs and limitations in order to encourage resource sharing and to reduce duplication of effort.

The goals and objectives of the tool environment are stated at the beginning of the following discussion. Section 4.2 specifies the concept of a model in the tool and defines the role of the selected notation of chapter 3 in a model. At this point all key software configuration items or products of the development process have been clearly identified. Section 4.3 describes the activities relevant to the tool in each phase of the scenario-based engineering process. Having defined the process and components an architectural framework remains to be developed in order to comply with the DSSA (domain specific software architecture) philosophy [Haddock, 93]. This is done in section 4.4. The architectural framework provides a strategy for developing an integrated environment, as well as a format for describing each necessary component of the tool environment. An object model and a scenario model and a scenario model based on the architectural framework is provided in the appendices.

## 4.1 Goals and Objectives

A tool, its architecture and components, as well as a development strategy using the tool will be developed in the following sections. The goals and objectives of developing the tool are listed.

- The tool must support as much of the entire software development lifecycle as possible.
- The tool must support multiple lifecycle approaches including the scenario-based engineering approach, the spiral model, a process model, etc.
- In supporting the scenario-based engineering process, the tool must utilize the concept of scenarios throughout the development effort by supporting development of scenarios for domain analysis and identification of responsibilities, refinement of scenarios to tasks, and use of scenarios as validation suites.
- The tool must automate and support management of components under development whenever possible. The management of software configuration items must be taken out of the hands of the developer.
- The tool must be aimed at developing massive software products that evolve over a period of time in a globally distributed, multi-version, and multi-user software development environment.
- The tool must be well integrated
- Limitations and weaknesses of the tools investigated must be overcome, and the strengths must be successfully replicated.

Several important characteristics, properties and features expected from a tool provide a guideline for development of a powerful and useful CASE environment [Wasserman, 87]:

- Underlying database: Data and tool integration is achieved with the use of underlying databases in a tool integration environment. A database also serves as a repository for software configuration items and other vital information.
- Integrated tool set: Information from one phase of development must be effectively mapped to other phases of development and all relationships between different analysis, design and implementation components must be automatically maintained.
- Rule checking mechanisms: The tools must include mechanisms that provide extensive support in tasks which are error prone and tedious. Semantic, syntactic and inter-component consistency and correctness must be ensured by the tool.
- Ease of use: The tool environment must be intuitively structured, have a consistent look and feel, and be easy to use. The learning curve must not be an obstacle to tool acceptance.
- Extensibility: An open architecture must be developed for the tool. Addition of new tools and integration must be facilitated by the architecture of the tool.
- Customizability: Current tools and components should also be modifiable to an extent. It must be possible to adjust and fine tune system configuration, access control and other options etc. to assist a developer or a team to suit their needs.
- Ease of navigation: The tool must effectively support access, navigation or traversal of design and implementation components in multiple ways and along multiple paths. This alleviates problems associated with management of complexity and also encourages reuse.

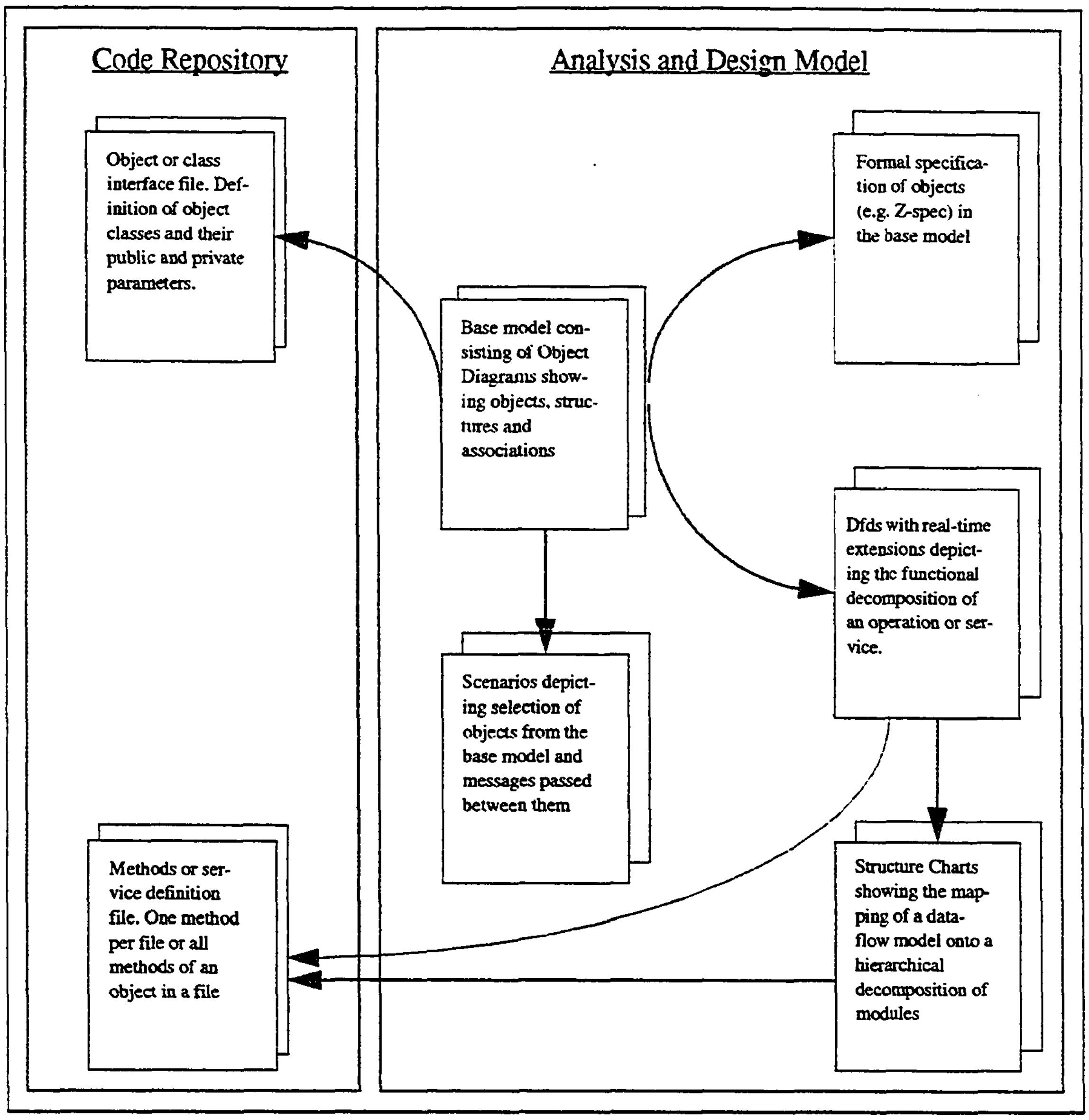

Code Repository
Analysis and Design Model
Object or class interface file. Definition of object classes and their public and private parameters.
Base model consisting of Object Diagrams showing objects, structures and associations
Formal specification of objects (e.g. Z-spec) in the base model
Dfds with real-time extensions depicting the functional decomposition of an operation or service.
Scenarios depicting selection of objects from the base model and messages passed between them
Methods or service definition file. One method per file or all methods of an object in a file
Structure Charts showing the mapping of a data-flow model onto a hierarchical decomposition of modules

**Figure 7. Relationship among components of a system under development**

This diagram shows the relationship among different components of development over the entire life cycle. A state chart or petri-net may also be associated with a scenario or an object. Additional modelling abstraction and methods can be added and related to any part of the model or the entire model, e.g. if flow charts are to be used to describe a method, then the diagram file containing the flow chart of a specific service should be related to the service in the object diagram, the Z-spec, or the file containing the method code. Each scenario may also be related to an implementation, or the file containing the code implementing the scenario. Each time an implementation using objects in the base model is made a scenario representation of the implementation may be added to the model, or vice versa.

## 4.2 Components of a Model

In order to develop specifications of a tool supporting the scenario-based engineering process, the system under development and the various software configuration items [Bersoff, 84] involved in the development process must be clearly specified. The notational and textual components of analysis, design and other life cycle activities are explained in chapter 3. A set of such components related to the same development effort and system is treated as a model which is described in this section. The software configuration items or components of the model may be further classified into 1) a base model, 2) a scenario model, 3) a functional model. Components maintained by the tool also include any other dynamic, behavioural, functional or static representation, formal specification, testing information and code. The relationship between the models is defined by the relationship between components of the notation shown in figure 2. Figure 7 shows another representation of the relationship among components of a system under development and figure 8 shows a refinement of figure 2 which includes models.

### 4.2.1 Base Model

The purpose of the base model is to depict the structural or static relationships between all the objects in the system. A base model consists of multiple pages, sheets or drawings of object diagrams. Each drawing may contain objects related by an aggregation or classification structure, or an association. An object may appear in multiple sheets but its characteristics must be the same. The implementation of this feature simply requires that multiple objects having the same name within a model, may not be allowed in the tool repository. This property is ensured by the tool and is required to be a standard feature. The base model may be defined as the union of objects, structures and associations in all the drawings.

The base model acts as a repository of objects. The repository of objects or the base model must maintain all relationships between objects as if they belong to the same set. As object models are refined the base model needs to be updated. For instance, there is no reason for maintaining an object model where no data types are identified when an implementation has already been delivered. However, options should exist in a tool to have multiple views of an object diagram. A form of view abstraction should be supported by the tool where either the object identifications alone, their attributes with or without data types, their services or all the characteristics of an object is selectively visible.

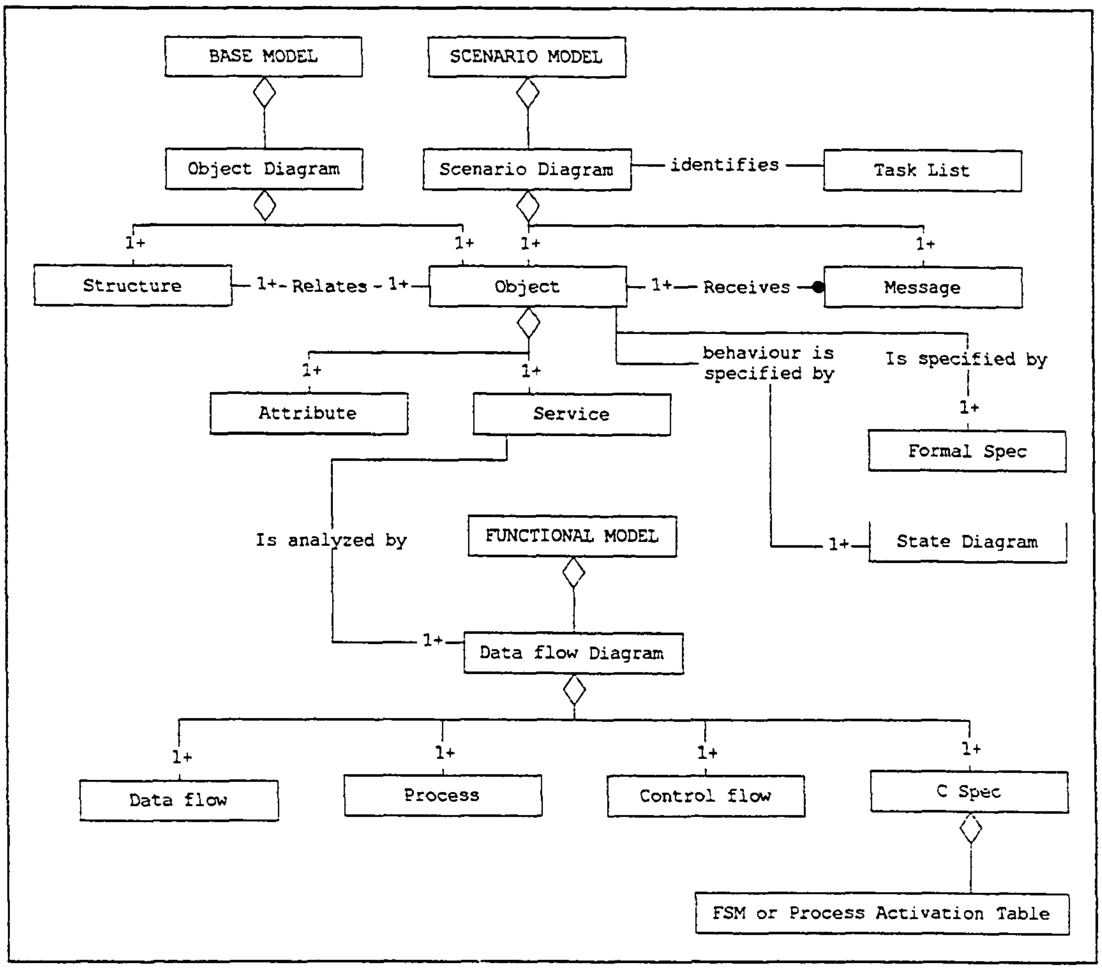


**Figure 8. Relationship between notation and model**

The relationship shown above is a refinement of figure 2 showing models, state diagram, task list and process activation tables. The base model, scenario model, functional model, their components and their relationships are specified. This can be used as the meta-model for the repository.

The base model is conceptually divided into two sections; the domain and the system. This distinction is made by the developer during or after object definition and the tool repository maintains this separation. No notational extensions to object diagrams are required. The purpose of this separation is to support the phases of development that follow analysis. However, during analysis this distinction is not very useful.

Additions to the base model are possible in different ways. An object or structure may be added to a sheet which does not already exist in the base model. If the object is present then the developer has the option to reuse it. All of the objects in the base model are available for use in the scenario model diagrams, and similarly an object may be added to the base model by defining it in a scenario diagram or copying it from another model into either the object diagrams or the scenario diagrams. When a new object not already defined in the base model is used in a scenario diagram, a new sheet is created in the object model containing this object. Structures or associations may be defined in the new sheet at any time. This allows a developer to start with scenario modeling or object modeling and to switch between the modeling techniques conveniently.

### 4.2.2 Scenario Model

The scenario model depicts the collection of various dynamic aspects of a system. It consists of multiple scenario diagrams. Three scenario diagrams exist for each scenario. This is based on the three levels of analysis discussed previously and explained by Haddock [Haddock, 93]. The mapping or relationship between the scenario diagrams in each set of three such diagrams is maintained by the tool. A scenario diagram in a scenario model has no direct mapping to any specific object diagram. Objects participating in a scenario may belong to one or many different object diagrams.

The environment layer and the domain layer scenarios are allowed to use objects from either the domain section or the system section of the base model. The technology layer is restricted to the use of system section objects. A user or developer is allowed to toggle between the different layers of the same scenario. This maintains the mapping between domain information, requirements, responsibilities and tasks. As explained earlier, the scenario model may not use objects not present in the repository. Addition of any such objects causes the addition of the object to the base model.

The number of possible scenario diagrams will grow combinatorially with the number of objects in the object model. Therefore, effective tools for reduction of duplication and inconsistencies should be provided. These tools provide facilities for identifying a matching scenario or a scenario which is a subset of another scenario.

Message parameters for each message in a scenario are maintained in the repository for provision to code generators in order to develop function prototypes, for consistency checking with the data flow diagrams of the functional model, and for developing test cases.

### 4.2.3 Functional Model

The functional model depicts the functional aspects of the system using data flow diagrams. The functional model may contain one or more data flow diagrams for each service of an object. The service or operation being analyzed appears as the process in the context data flow diagram. This diagram may be decomposed to several levels. Multiple sets of data flow diagram are supported for each operation in order to represent and understand polymorphic services. The mapping between the service and its data flow diagrams are also maintained by the tool.

If the scenario model is developed prior to the development of the functional model, then certain information may be carried over to the functional model. A data flow shown in any scenario along with the invocation of a service implies that this flow must be present in the context diagram of the service. A context diagram for each such service should be generated by the tool in order to assist the developer in maintaining consistency. If a polymorphic service is being invoked than the consistency between scenario model flows and functional model flows is based on the union of flows in the two or more data flow diagrams.

### 4.2.4 Behavioural Model

The behavioural model as stated in the notation is optional. This is due to the fact that the scenario models provide the dynamic aspects of a system, relevant to its responsibilities and expected capabilities. However, state-transition diagrams or nested state diagrams may further illustrate behavioural aspects of each object. This may be vital to certain development efforts.

The behavioural model consists of state transition diagrams for each object. The use of nested state diagrams for such representation has also been presented and discussed by Rumbaugh et al in the explanation of their dynamic model [Rumbaugh, 91], and Shlaer et al [Shlaer, 92] in their book on modeling object states. A transition occurs every time a message is received by an object. The collection of the various states of an object and the transitions due to messages in different scenario diagrams can be combined into a state diagram or nested state diagram of the object. Again consistency between the messages which are events of the state diagrams and messages arriving at objects in different scenario diagrams can be checked by the tool.

### 4.2.5 Formal Specifications

Formal specifications provide 1) a consistent form for the expression of specifications, an axiomatic mechanism to support expression of relations among specifications and 3) a proof theory which supports mathematical analysis of specifications [Spivey, 92]. As discussed in the previous section a set of formal specifications will possibly be related to an object. A formal specification model will consist of a set of such sets. The formal specification for each object may be maintained in a text file, and hence the formal model will contain at least as many text files as there are objects in the base model. The mapping or relation between formal specification text files and objects must also be maintained by the tool.

## 4.3 The Scenario-based Development Process

A development process requires a cohesive plan that includes methods, procedures, tools, documentation, quality assurance procedures, control mechanisms and management planning to ensure the development of a system with conceptual integrity. Although not explicitly addressed by most life cycle models reusability must be a factor in all the phases of such a development process in order to maximize potential gain. Figure 9 provides the life cycle for the scenario-based engineering process as shown by Haddock [Haddock, 93]. Figure 10.shows an interpretation of a development process using the scenario-based engineering process and the tool in the spiral model of development and enhancement [Boehm, 88].

The phases in the scenario-based engineering process are "analysis, component selection, system design and component modification, implementation, integration, followed by archival of new components" [Haddock, 93]. There are multiple feedback loops between phases and the development can be seen as a process model rather than a sequential development strategy. The phases are essentially stages of refinement and the entire model develops as work progresses. The individual tasks are not the main focus.

The following sections attempt to define the different phases of development using the scenario-based engineering process with respect to the proposed tool. This discussion ties together the notation and models described in the previous sections and the explanation of the development process by Haddock [Haddock, 93]. A basic assumption of the discussion at this point is the availability of the tools or tool to support the process. Therefore, it provides insight into the functionalities of the tool and the role of the models described earlier.

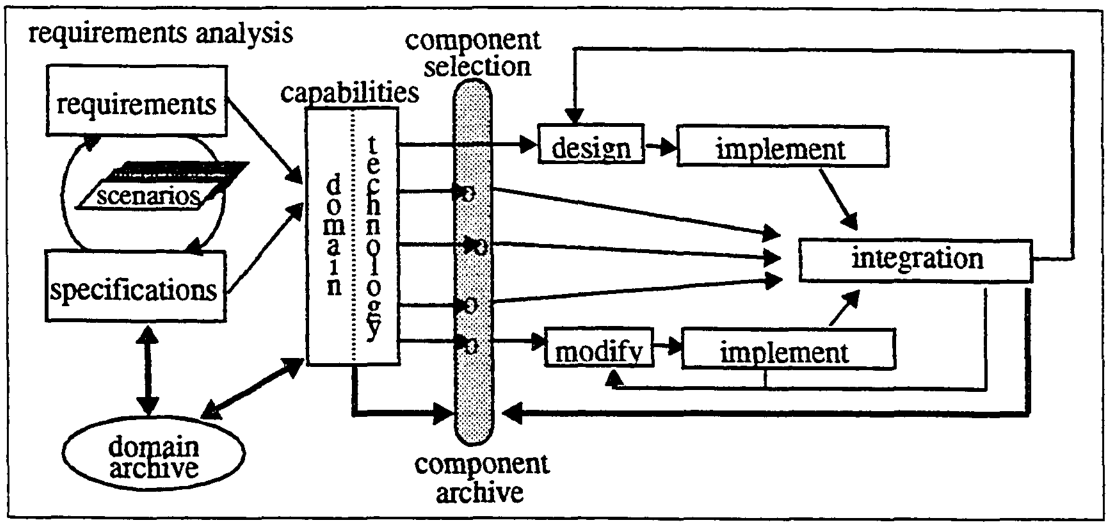


**Figure 9. Scenario-based Engineering Process: complete life cycle**

The life cycle of the scenario-based engineering process is used from the thesis by Haddock [Haddock, 93]

### 4.3.1 Requirements Analysis

Domain analysis is performed with the use of object diagrams and environment layer scenarios. Scenarios exist in three layers: environment, domain and technology [Haddock, 93]. The role played by scenarios in this phase is to identify responsibilities and requirements and to verify if they are being met by the components of an object model. Scenarios are later refined into task lists and tasks are further refined and analyzed into subtasks etc. The refinement of environment layer scenarios lead to domain layer scenarios and tasks map to capabilities required of the system. The domain layer scenarios are finally transformed or mapped to technology layer scenarios, but this requires the presence of implemented components of the system. These scenarios play an important role in component selection or design specification.

| **Planning** | **Risk Analysis** |
|---|---|
| 1. Requirements gathering and project planning.<br>5. Further requirements gathering based on customer evaluation.<br>Plan reference architecture.<br>9. Plan design architecture, divide design among groups or individuals, and assign responsibilities.<br>13. Review models resolve group interaction and interface issues.<br>17. Review code and test results developed for object model. Develop strategies for final implementation.<br>21. Develop plan for testing, integration etc. | 2. Risk analysis based on initial requirements.<br>6. Risk analysis based on customer reaction on initial model.<br>10. Risk analysis of design objectives and assignments.<br>14. Risk analysis based on design and formal schemas.<br>18. Risk analysis based on test results and formal schemas.<br>22. Review final plans. |
| **Customer Evaluation** | **Engineering** |
| 4. Customer evaluation of analysis model.<br>8. Customer evaluation of design additions.<br>12. Customer evaluation.<br>16. Customer evaluation.<br>20. Customer evaluation.<br>24. Customer evaluation. | 3. Developing the base model to analyze the domain. Identification of objects, attributes, services, structures and associations. Identification of relevant scenarios.<br>7. Modifying base model and scenarios based on evaluation and risk analysis. Develop design scenarios. Develop dfds for each service. Develop state charts for objects.<br>11. Make necessary additions and modifications to model. Develop formal schemas. Develop structure charts from dfds.<br>15. Select components. Develop code for object models. Test.<br>19. Develop code for implementation of scenarios, using code from step 15.<br>23. Integration, testing, etc. |

**Figure 10. Scenario-based development in the spiral model(Boehm, B)**

A possible lifecycle for scenario-based development using proposed tool and design method. Can also be viewed as a method for managing the development process. The spiral begins at planning step 1 continues clockwise through the numerical order. Although 6 major iterations are shown further iterations and various variations are possible. The scenarios are important in every spiral cycle, especially in the engineering phase. In engineering step 6 the scenarios are used as validation suites for both white-box or structural and black-box testing. In the risk analysis phase of earlier iteration one could make the go/no go decision based on negotiations such as selectively choosing scenarios to implement.

Both the base object model and the scenario model are developed concurrently. Structures and associations between objects may become obvious at any time during development. Attributes or services may be identified during development of the base model or the scenario model. Any objects and structures in the base model are finally used in scenarios, and similarly an unidentified object may be added into the object model through a scenario. Therefore the two models grow during development and the developer is not locked into the specific development of one model or the other. The tool supports such a development process by maintaining in the repository, the relationship between the two models. Objects may be added or refined from either model. The functional model is also developed concurrently with scenarios and object diagrams when a specific service is analyzed in detail.

### 4.3.2 Component Selection and System Design

Component selection entails selection of objects from the object model and corresponding implemented components. The criteria for component selection are based on meeting the capability requirements derived from the scenarios. The component selection process is followed by system design and component modification. System design comprises of implementation strategies of the object model to make components available for use. Scenarios in the technology layer define the behaviour of components of the system, which are implemented by components available in the archive. The object model and object diagrams provide the access into this archive. The data flow diagrams and structure charts, if any, are used as an aid for detailed design issues such as identification and analysis of functionalities of services provided by objects and invoked in scenarios. State diagrams may also be developed for each object to better understand the behavioural aspects of each object.

### 4.3.3 Further System Design and Component Modification

The primary guiding mechanism in system design are all the models which are developed. The object model provides the details pertinent to each object, while the scenario model act as a guide to their functionality and hence provide strategies for component modification. Some basic techniques for modification are use of aggregation or inheritance not already identified. Data flow diagrams or structure charts in the functional model are the design documents for each service or operation. Mapping strategies from such diagrams to code are well understood and extensively discussed in the literature for the functional paradigm of software development. These models are intuitively obvious and simple to understand, as the scenarios provide further insight into each services role in the system.

Data types of object attributes, messages and values returned have to be identified in this phase. The object model and scenario models must be checked for completeness. Objects must be categorized as active or passive. The constraints on services have to defined and control flows and data flows appearing as messages in the scenario model need to be refined or decomposed in the functional model. Control specs must also be developed including the corresponding process activation charts or decision tables. Structure charts and state diagrams may also be developed based on needs.

**4.3.4 Implementation**

In this phase the objects in the object model have to be implemented as abstract data types, where there static features are specified by their structures and characteristics. When all the necessary objects are available for use, implementation of components of the system which satisfy the identified capabilities and responsibilities can begin. All the models again play a major role. The functional model works as an implementation guide for each service.

The scenarios also provide a verification and validation mechanism for the entire development process. At any stage of analysis, design or component selection the scenarios work as validation suites. The scenarios can also be to develop effective test cases. The formal specifications for each object also provide a means of validation using proofs. Formal proofs can be made using the formal specifications which can be developed throughout the analysis, design, component selection or implementation phases. These can be valuable for enhanced user confidence in safety critical systems.

With the availability of the three completed models, code generators can provide valuable automation support. Our experience with code generators is that skeletal code frames are generated by most generators. However, generators can be built which use information from all the models including formal specifications to generate a robust code frame, complete with object interface definitions, function prototypes, data structures and code file organizations and structures. A permanent link between generated code and design documents could be maintained by the tool.

### 4.3.5 Integration

Integration tasks are never simple or predictable. The key to better integration is detail design of interfaces between components. Object oriented techniques assist in achieving well defined interfaces. An abstract approach to the combination of models which contain design documents and code can be to treat a model itself as an object. The objects and structures can be treated as attributes, while the scenarios are services. This is a conceptual perspective and not a design strategy. It is also difficult to formalize.

However, integration is aided greatly by the automated management of relationships between design and analysis documents, requirements, and code. Once an integration strategy based on the reference architecture is developed, the tool can be used to modify the models and to combine them. Scenarios can also be developed just to study the integration of two or more developed components within the same model or in separate models.

### 4.3.6 Archival

The goal of reuse as well as version management and other tasks depends on the proper archival of objects and models developed. Archival pertains to the storing, identification and retrieval features provided by tool. The mapping between a developed system, its components, and design documents needs to be maintained after completion and throughout a project. Reuse can be achieved at various levels of a model or a developed system. The entire system can be reused or small components within the model can be reused. The tool must support this at all levels. Reuse must also be supported for a set of interrelated components and design documents. This can be achieved by explicit copying and moving mechanisms.

Management of an archive of models is also an important issue. Configuration management tasks do not end with delivery, but continue throughout in the form of archive management. Access to models or parts of models, updating or adding functionalities to the models, correcting them or reorganizing them is an ongoing process, which must also be supported by the tool. Facilities to search through an archive using multiple paths for identification of reuse components must also be provided.

## 4.4 Architectural Framework for the Tool

A tool architecture is presented in this section based on the experiences from chapter 2, the notation from chapter 3 and the process discussed in the previous sections. The tool architecture does not differ greatly from the tools surveyed. Figure 11 specifies an integration framework and tool architecture explicitly for the tool to support the scenario-based engineering process. The following sections elaborate on the different components and layers identified in the architecture. The interfaces or communication links among the different tools and layers must be well defined. The tool environment must be well integrated.

### 4.4.1 User Interface

The user interface layer provides a common user interface with the use of a standardized interface toolkit e.g X-Window System, Motif, etc. Such a toolkit provides consistent and standardized communication between the interface and the tool management services, browsers and editors. A well defined presentation protocol is followed to ensure consistent look and feel of the tool. Windows, icons, menus, prompters and common mouse operations are also ensured by this layer. The primary function of this layer is to provide presentation integration [Thomas, 92].

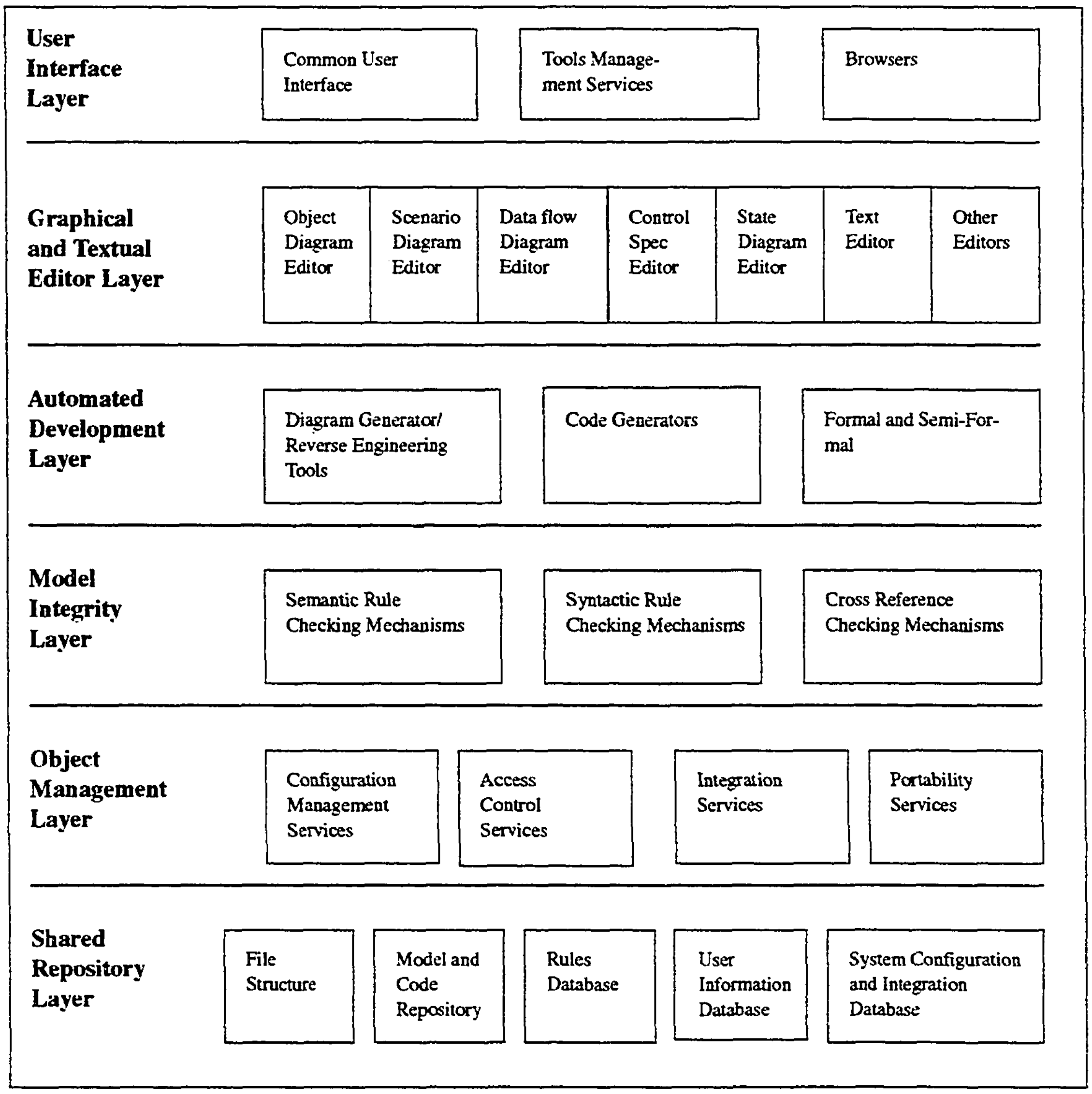


**Figure 11. An architectural framework for SEP tool integration**

A tool management service is also provided in this layer to manage invocations of different editors, code generators, reverse engineering tools and rule checking mechanisms. The tool management service supports initiation of edit sessions from a common menu or from selection within an editor of an element of a notation. The tool management services also provide specialized services for interoperability, information exchange between tools and repository, and common tool access. Security functions may also be handled by this layer [Pressman, 92].

Hypertext links [Cybulski, 92] and data dictionary based navigation are supported by the browser facility in this layer. The multipath links supported by the browser are between the following corresponding items:

- objects in the object diagrams and objects in the scenario diagrams
- objects in the object diagrams and state diagrams
- objects in the object diagram and formal or semi-formal specifications
- objects in the object diagram and object interface definitions (code)
- objects in multiple object diagrams or scenario diagrams
- the three layers of a scenario
- the scenarios and task lists
- services of an object and data flow diagrams or control flow diagrams
- services of an object and method files (code)
- services of an object and structure charts
- data flow diagram and structure charts

### 4.4.2 Graphical and Textual Editors

The tool environment must provide editors for each modeling technique or method used in the scenario-based engineering process. At least three different graphic editors are provided for this purpose: 1) object diagram editor, 2) scenario diagram editor, 3) data flow diagram editor. Each editor supports the notation presented in chapter 3 and has well defined hooks for integration and data interchange interfaces for communication to the repository or underlying layers. The addition of any of the notational graphical elements and associated textual information leads to automatic addition of the information to the database. The editors support dragging or moving items within a diagram. A message arrow, a data flow or a structure is not lost by moving an object or a process.

Each presentation appears in a separate window with menus and/or palettes supporting the features of the notation. The user may invoke any of the editors at any time and the system supports invocation of one editor from another. This function is actually performed by the tools management services. For instance a data flow diagram edit session can be invoked by making the correct menu selection from a menu which appears on clicking on a service. If a data flow diagram exists the editor accesses the database and recreates the diagram. If a diagram does not exist then a new edit session is initiated.

Text editors with hypertext connections to the graphic editors are also provided. Text editors specialized or configured to support formal specifications, data dictionary support or coding are also provided. Additional text editors are provided for making annotations and textual explanations to diagrams. Specialized editors for updating diagram components such as object specific editors, control specification editors, etc. are also provided. Each of these editors can be invoked from the diagrammatic components with which they are related.

### 4.4.3 Automated Development

The automated development layer provides support for automatic code generation, reverse engineering, and formal or semi-formal text generation. Specific code generators for different languages using as much information from the model as possible are provided. Completeness of model is not a major criteria. However, in order to generate code skeletons, formal specification skeletons, or semi-formal texts such as the object description skeletons (ODS) used in HOOD, model consistency and correctness are required. Once code is generated it is stored in files in a file structure and the database, so that browse features can be used.

Tools are also provided for reverse engineering and re-engineering tasks. Diagrams or formal specifications can be developed from existing code using these tools. Some manual reorganization of existing source code into different files may be required for these services to operate correctly. Diagrams may even be generated from other diagrams, as discussed earlier e.g. context data flow diagram of a service from the scenario model.

Code, whether it is generated by the tool or not, also falls within the scope of the tools management. Although, most code generators generate code and from then on have no connection with the graphical design objects from which they are mapped, this tool environment must store the code in the repository or an associated file structure. If the code is changed considerably then the tool should monitor this change. The reverse engineering tools can be used to check for conceptual integrity between the code and the design documents.

### 4.4.4 Model Integrity

This layer provides an array of rule checkers for notational correctness and consistency. These features can be invoked when the developer is ready and are not active constantly. This allows the developer to save an inconsistent model during different stages of development. The rule checking mechanisms check for syntactic, semantic and heuristic integrity of the model components. Consistency between different components of the model can also be ensured by the use of cross reference checking mechanisms. These tools also use the tools management layer to interact with the repository when needed.

### 4.4.5 Object Management

The object management layer plays a major role in integration of different tools and model components by providing a wide array of service. The objects referred to are process outputs and not objects which are parts of models. The configuration management services support various configuration control functions such as version control, maintenance of relationships between components, copying and moving features within models, and multiple model integration. Access control functions handle problems associated with multiple users or multiple development teams. Access control can be specified based on read, write and execute criteria between individuals groups or all users similar to file access rights in Unix.

Integration services are a set of functions which tie together everything above the object management layer (editors, generators, rule checkers) with the shared repository layer. Separate integration facilities allow modifications in the tools integration framework. Portability services refer to the conversion of models into formats such as CDIF (CASE data interchange format). These services also provide access to the database files which can be transported between different machines within the same tool.

### 4.4.6 Repository

The shared repository layer consists of multiple databases where all information pertinent to tool operation and all software configuration items are stored. The repository is important in order to achieve data integration. The repository or database must provide metamodel, query, view and data interchange services [Chen, 92]. The repository also provides its own access control mechanisms.

A separate database is provided for source code and for the models. The developed source code and models are actually kept in files to which the database contains pointers. An associated file structure which may be user defined can be used for storage of these files.

The underlying database can be an object oriented or relational database. The data model, entity relationship schema, or relational schema for storage of software configuration items can be easily derived from figure 2 or figure 8. A corresponding file structure which may be defined by the user can contain the different target files.

A database is dedicated to access and storage of rules used by the rule checking mechanisms. This allows modification of the rules and customizability to a certain extent. The tool or system configuration information, integration information, information regarding the file structure and information regarding different user access rights and privileges are also kept in a separate database.

## 4.5 Reuse

Reuse is a key concept in the DSSA philosophy [Mettala, 93] and in the scenario-based engineering process. Not only does the tool support reuse at different levels, but the tool architecture itself is open and available to reuse. One of the primary reuse related tasks is the identification of reusable components. The tool environment described above supports the identification of developed components and design documents for reuse. The graphical access to developed components through models is a powerful tool for quickly understanding and identifying a reusable component. Scenarios are effective tools for quickly comprehending the dynamic behaviour of the system. Objects also inherently support reuse by providing a well defined interface. The relationship between the object model, scenario model and the functional model, and its dynamic support by the editors and browsers make this environment extremely conducive to searches for reusable components.

Once a component is identified, the reuse of a component and all other related components is possible because the mapping between different software configuration items is rigorously maintained. Reuse is possible at various levels or granularity. At the most basic level a single component of a model such as a diagram or an object from the base model may be reused. At a higher level of granularity an entire scenario, the participating objects, the source code, etc. together may be reused. The tool provides specific copying and moving facilities for such reuse. These utilities are provided primarily by the user interface layer.

The modification of a component to be able to effectively reuse it and to develop components which are especially suited to reuse is a task which the tool cannot directly support. These activities are achieved by the proper use of design concepts of the object oriented paradigm and the scenario-based engineering process. However, the tool may make these tasks relatively easier due to its tight integration and comfortable user interface.

## 4.6 Tool Integration

Tool integration implies that tools "function as members of a coherent whole" [Thomas, 92]. Four kinds of tool relationships which define integration are: 1) Presentation integration, 2) Data integration, 3) Control integration and 4) Process integration. Presentation integration refers to a tool's ease of use and the reduction of the developer's "cognitive load" [Thomas, 92]. Data integration attempts to ensure that all information in the environment is consistent. Control integration refers to the flexibility and power of combining multiple tools in a single environment as needed by the development process. Process integration pertains to a development process and how well it is supported by a set of tools.

The previous discussion in section 4.3 focuses on process integration assuming use of the scenario-based engineering process. Data integration in the proposed architectural framework (figure 11) is achieved by the use of the shared repository, and the functions of the object management layer. Control integration is supported by well defined communication links, the object management layer and the tools management services. Presentation integration is the job of the user interface layer.

Vertical or full lifecycle integration is consistency of information produced in various phases of a development process. This is achieved by the object management layer and the automated development layer. Horizontal or methodological integration means integrity of design information within a phase of development. This is achieved by the shared repository layer, the model integrity layer, and the combined use of the user interface layer and the editors.

# 5 CONCLUSION

This thesis begins by presenting the observations and experiences from development efforts on six different CASE tools supporting various object oriented methods. The tools surveyed where found to be effective only in specific phases of software development. None of the tools provided a well integrated environment for software development. Although Cadre Technology's Teamwork provided the largest array of tools, the analysis and design methods supported were cumbersome and unpopular. The other tools were good analysis or design tools but they failed to support the entire development life cycle. However, many important lessons were learned from the experience of using the tools.

One of the main goals of this thesis was the specification of a tool to support the scenario-based engineering process. The architectural framework and the use of the tool in each phase of development partially achieve this goal. This thesis also aspires to define a domain specific software architecture (DSSA) specification for the SEP tool. The scenario-based engineering process itself is used to achieve this goal. An object model (appendix A) and a scenario model (appendix B) of key activities is developed.

A set of notational components is identified to support software development using the scenario-based engineering process. The use of the components in the development process is explained. The key software configuration items are identified. The tool is explicitly specified to support development using the scenario-based engineering process. Activities and tool support for each phase of the scenario-based engineering process are addressed.

The tool and the architectural framework defined in this thesis provides an integrated environment for effective use of the scenario-based engineering process to build, maintain and reuse quality software. The tool specified supports the entire life cycle of the SEP. The tool environment provides a collection of effective tools which are tightly integrated. Graphical tools are provided for specialized support of domain and requirements analysis, component selection and design. Automated support is made available for implementation and integration tasks. Extensive rule checking mechanisms are provided for semantic and syntactic integrity of models and systems developed using the tool. Reverse engineering tools and code generators are also supplied.

The tool environment is developed upon a shared repository and has a common user interface layer on top to give it a consistent look and feel. The configuration management tasks for managing the scenario-based development process is automated and supported by the tool throughout the life cycle. Reusability of developed components and development of reusable components are enhanced by the tool environment and by the facilities available to the developer. Traceability of development components to requirements documents and persistence of relationships between various components are also effectively supported by the tool environment.

# 6 FUTURE WORK

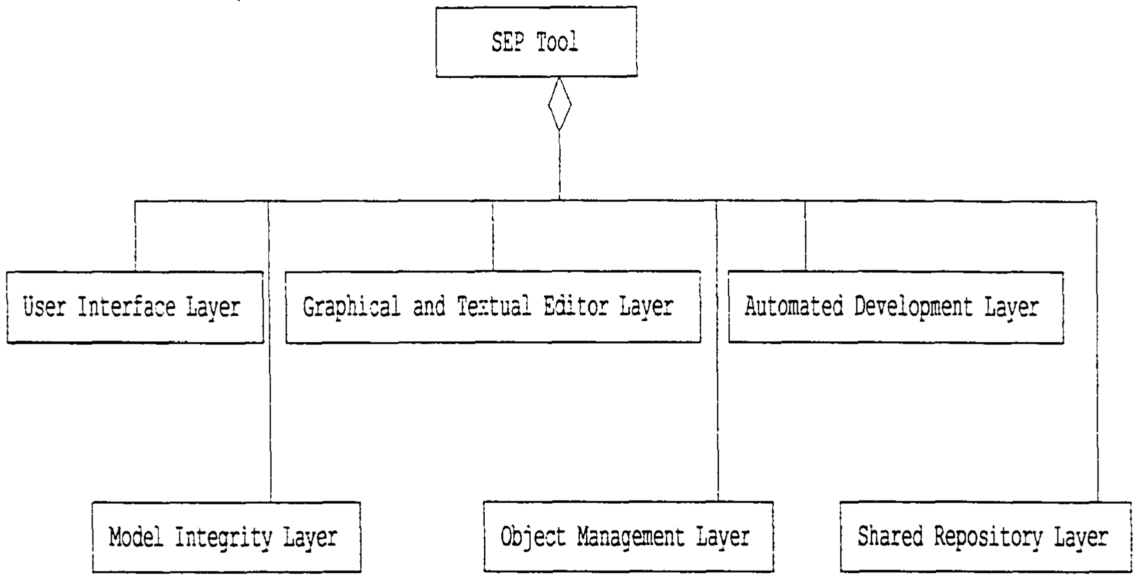
One obvious goal that can be achieved in the future is the implementation of the tool environment. This task can be achieved by developing each component separately. The scenario-based engineering process should be used in the development of the tool. Some of the components are straight forward and can be developed quickly. However, some of the components such as the object management layer facilities will be a greater challenge.

We are currently assessing configurable graphic editors to be able to develop some of the editors described in this thesis. We are also looking at mechanisms and languages which can be used in association with the editors to provide consistency. The rules for consistency and correctness must be clearly specified before this task can be undertaken. The code generators and reverse engineering tools can also be developed in a completely isolated manner as long as the interfaces to the database or the editors are clearly defined.

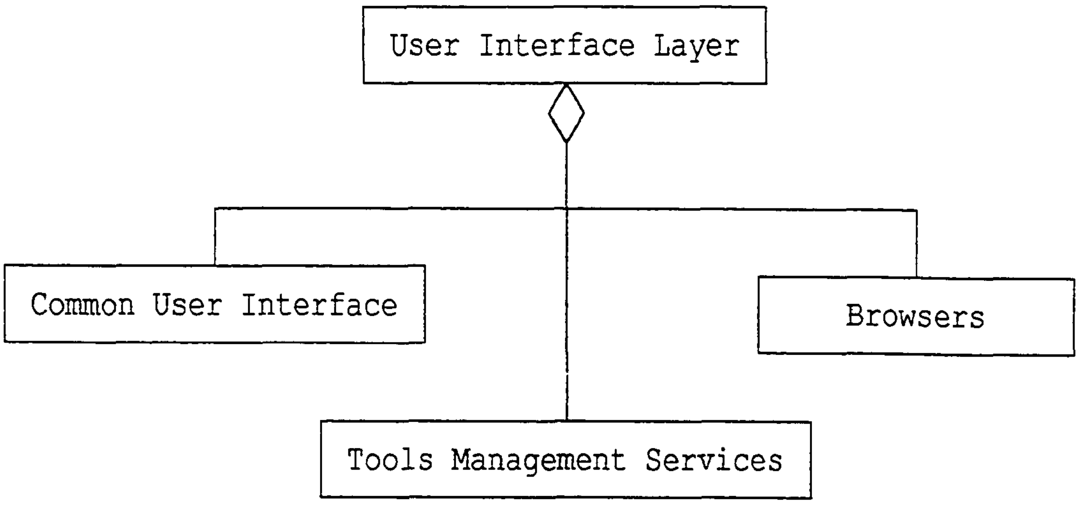
The repository is also an item which can be handled with relative ease in the near future. The availability of object oriented databases and powerful relational databases will greatly help such a task. The data model of the database is already available in this thesis.

# APPENDIX A

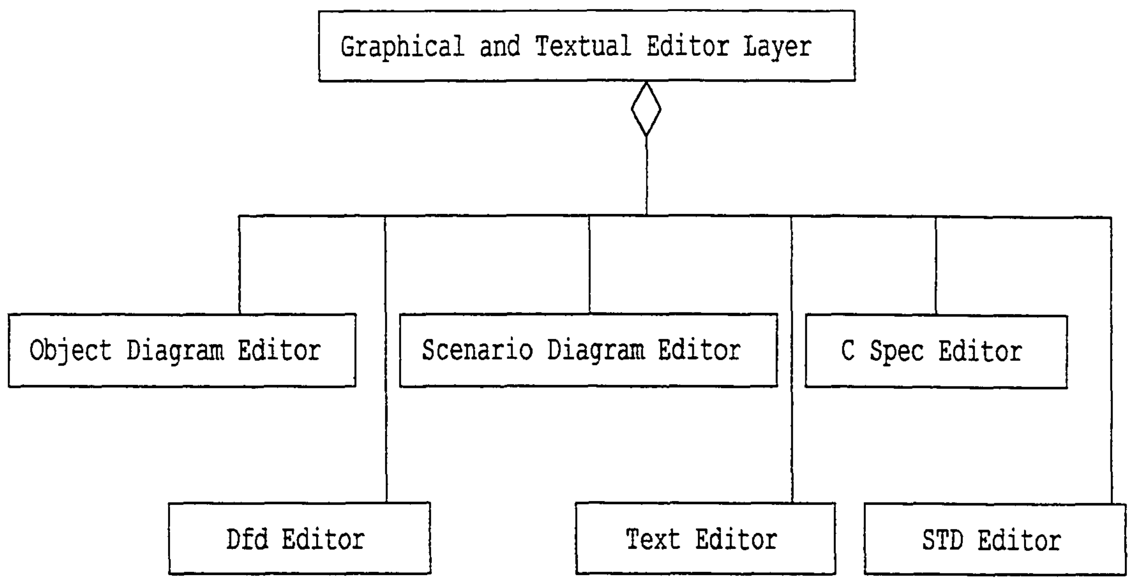
## OBJECT MODEL OF THE TOOL FOR THE SCENARIO-BASED ENGINEERING PROCESS



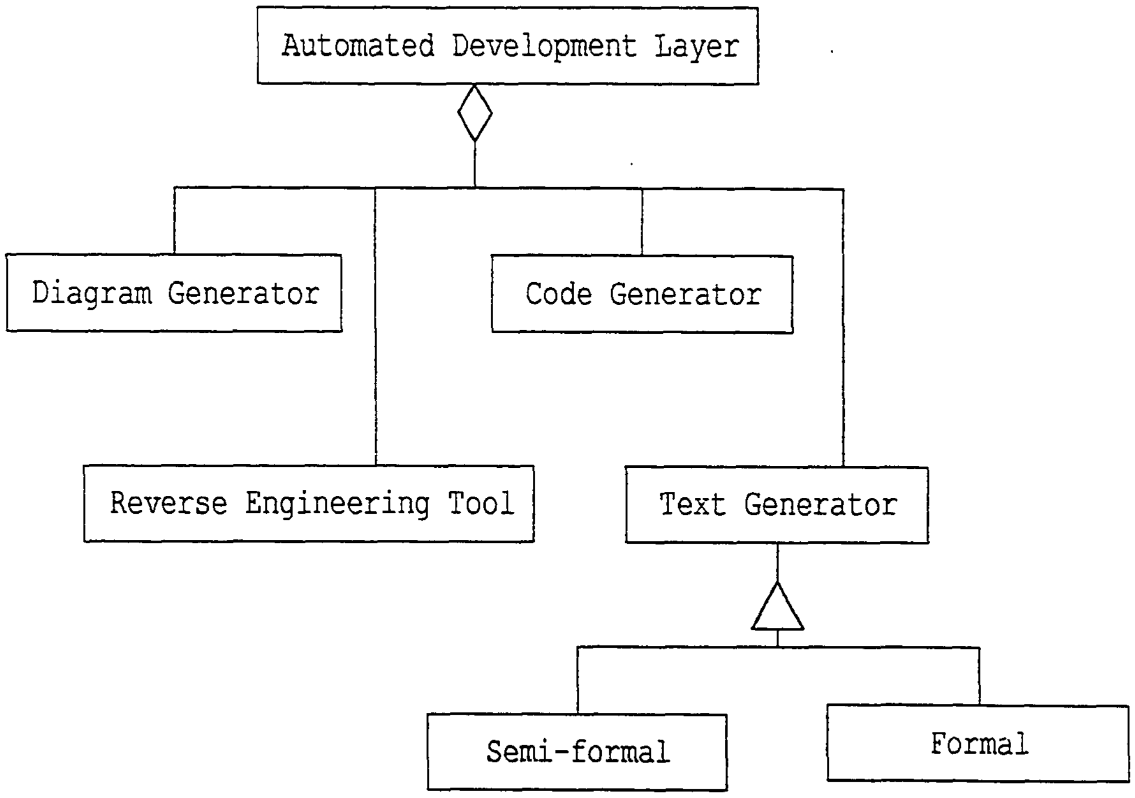
<u>Object Model of the SEP Tool</u>



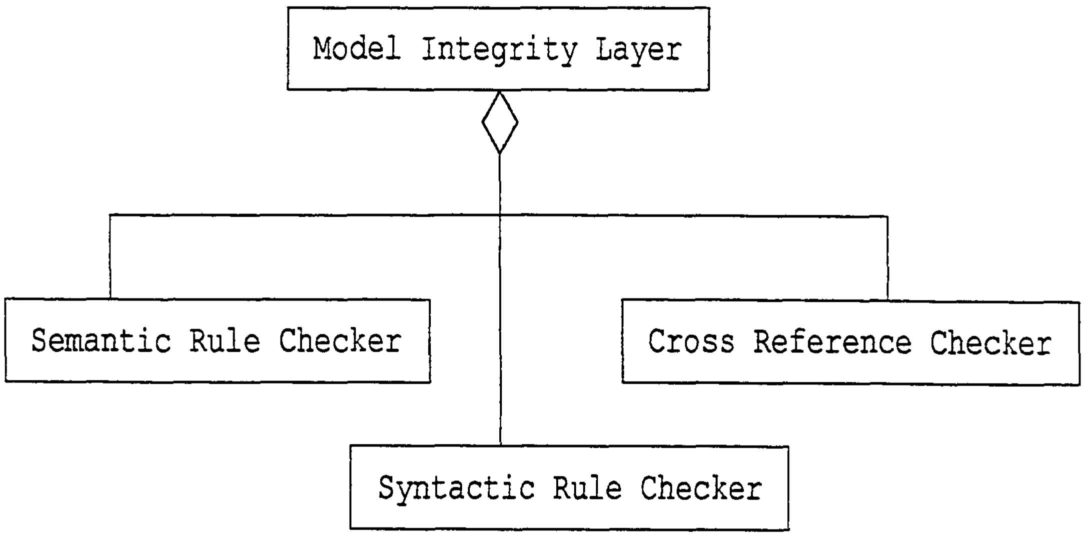
Object Model of the User Interface Layer



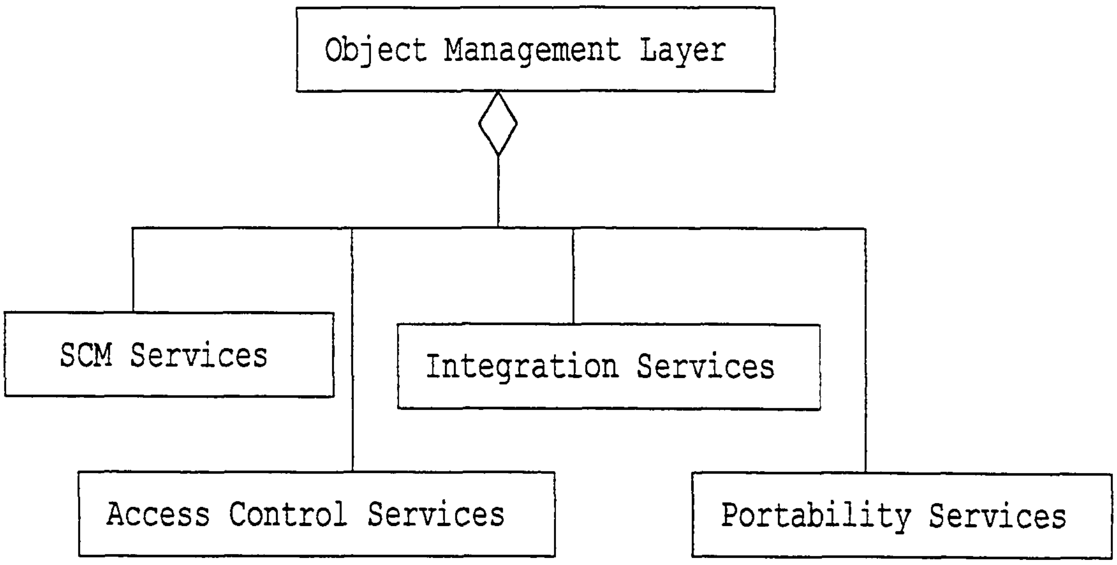
Object Model of the Graphical and Textual Editor Layer



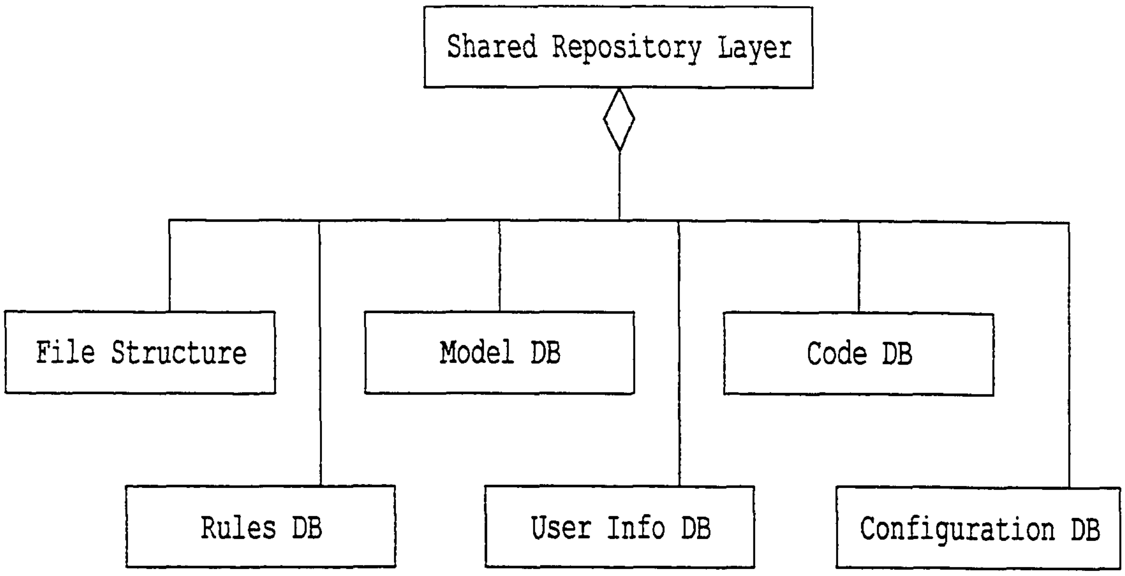
Object Model of the Automated Development Layer



Object Model of the Graphical and Textual Editor Layer



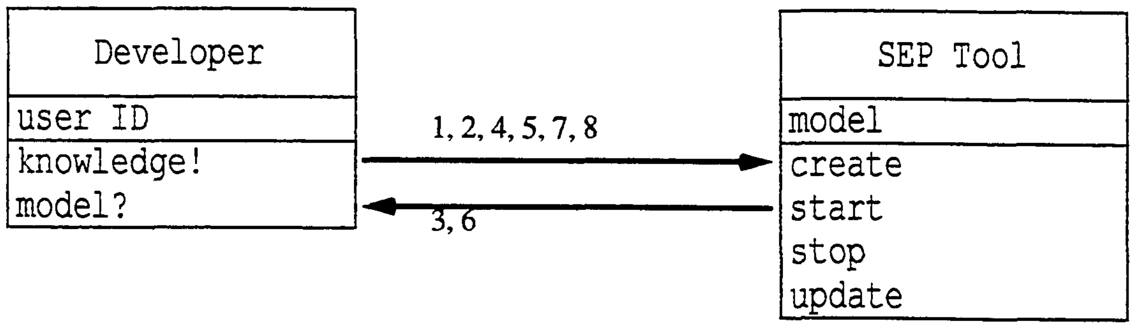
Object Model of the Object Management Layer



Object Model of the Shared Repository Layer

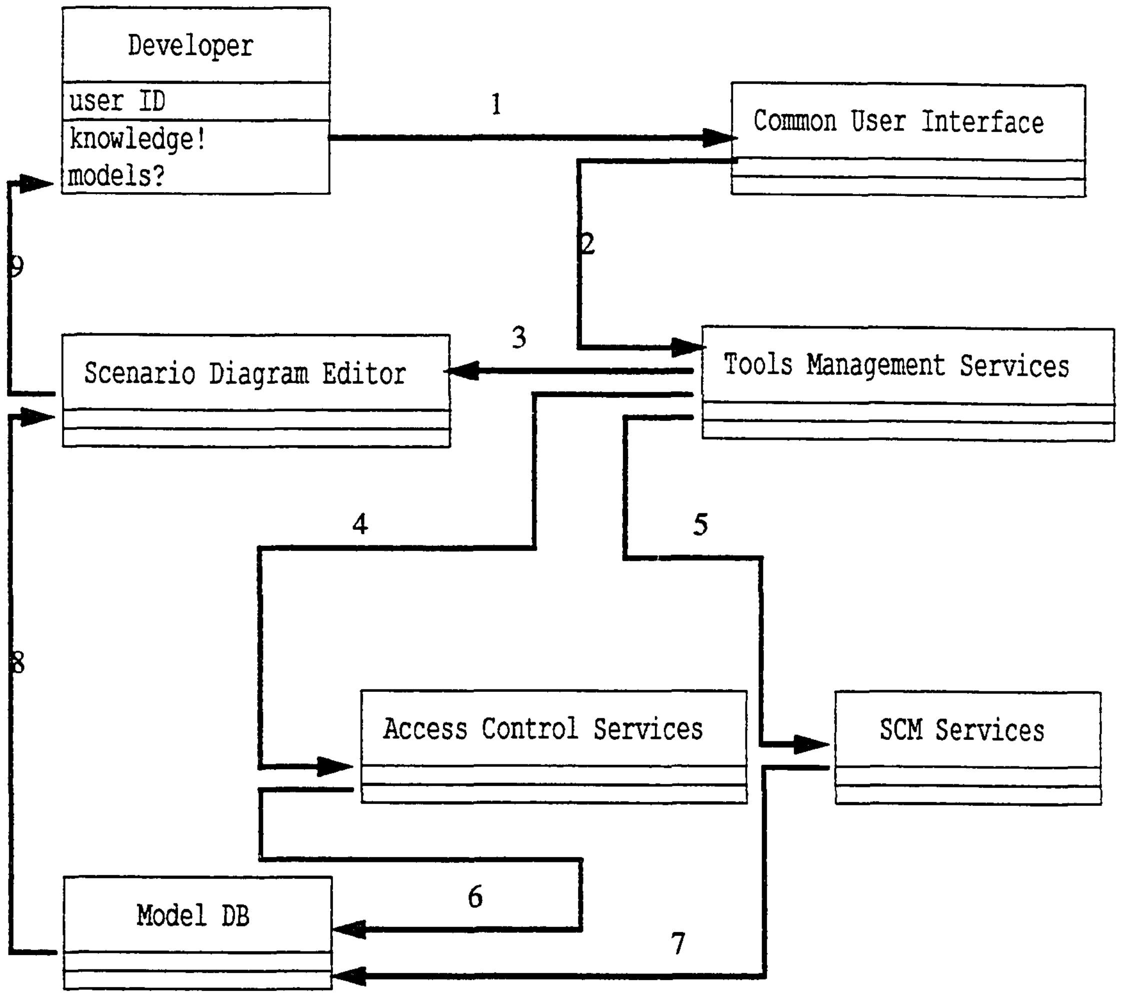
## APPENDIX B

## SCENARIO MODEL OF THE TOOL FOR THE SCENARIO-BASED ENGINEERING PROCESS



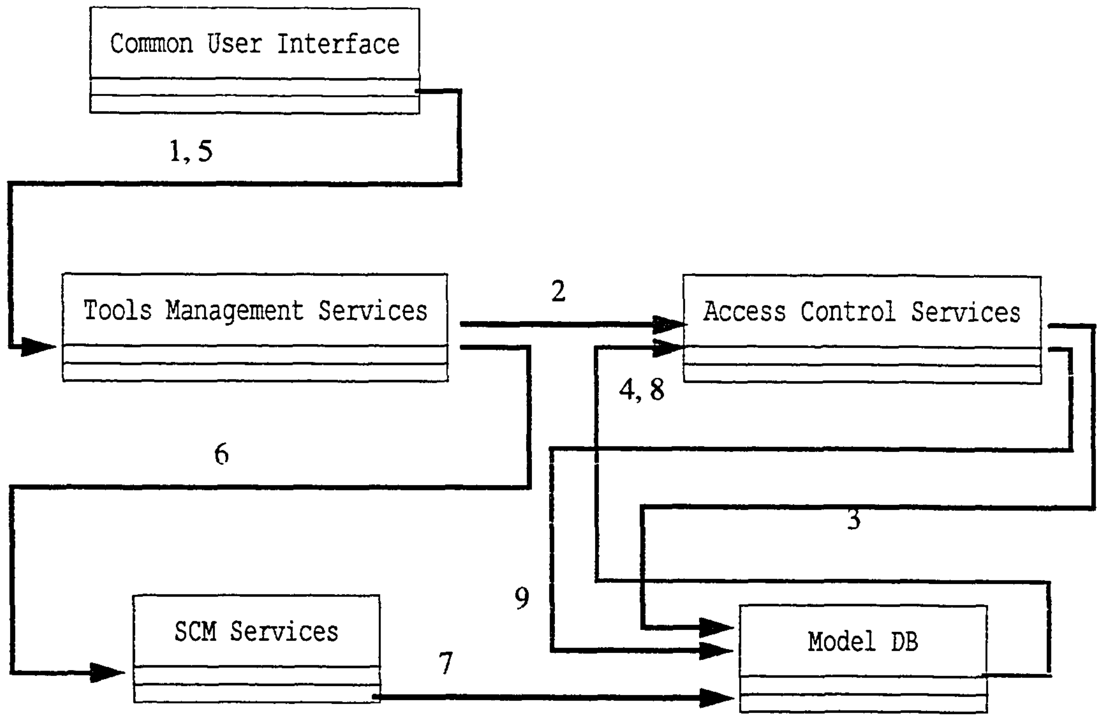
Scenario 1: Level 0 : Developer Creates Model

1. Developer starts tool
2. Developer creates new model or selects model
3. Tool returns model
4. Developer invokes specific tool or editor
5. Developer selects file, drawing, etc.
6. Tool returns selection
7. Developer updates model
8. Developer stops tool



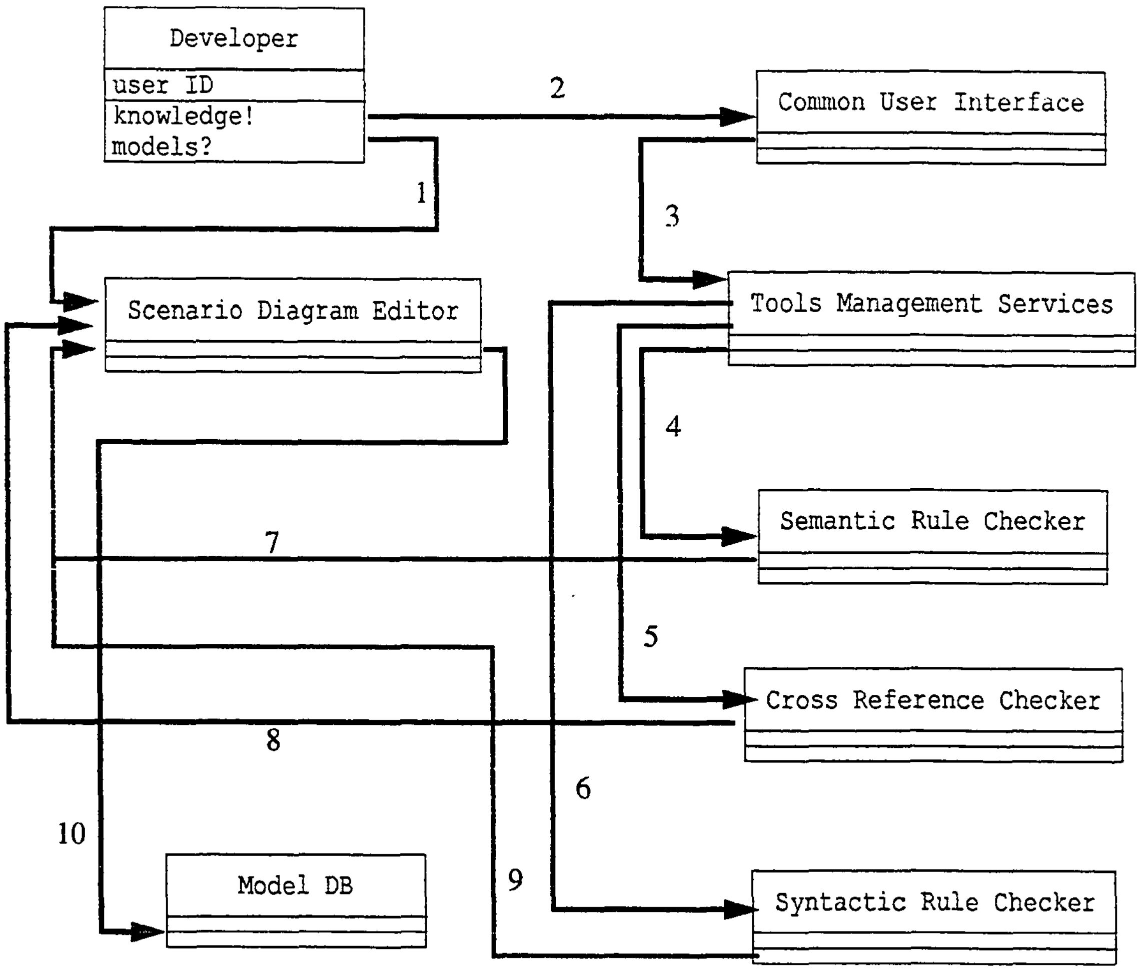
Scenario 1.1 : Level 1 : A model diagram is provided

1. Developer selects a model using the common user interface.
2. The common user interface passes the information to the tools management services.
3. The tools management services invoke the editor
4. The model ID and the developer ID are passed to the access control services.
5. The key to the model and scenario requested is sent to the SCM services.
6. If the user has access authorization then an OK is sent to the model DB.
7. An appropriate query is sent to the model DB.
8. The scenario diagram information is sent to the scenario diagram editor.
9. The developer can update the scenario.



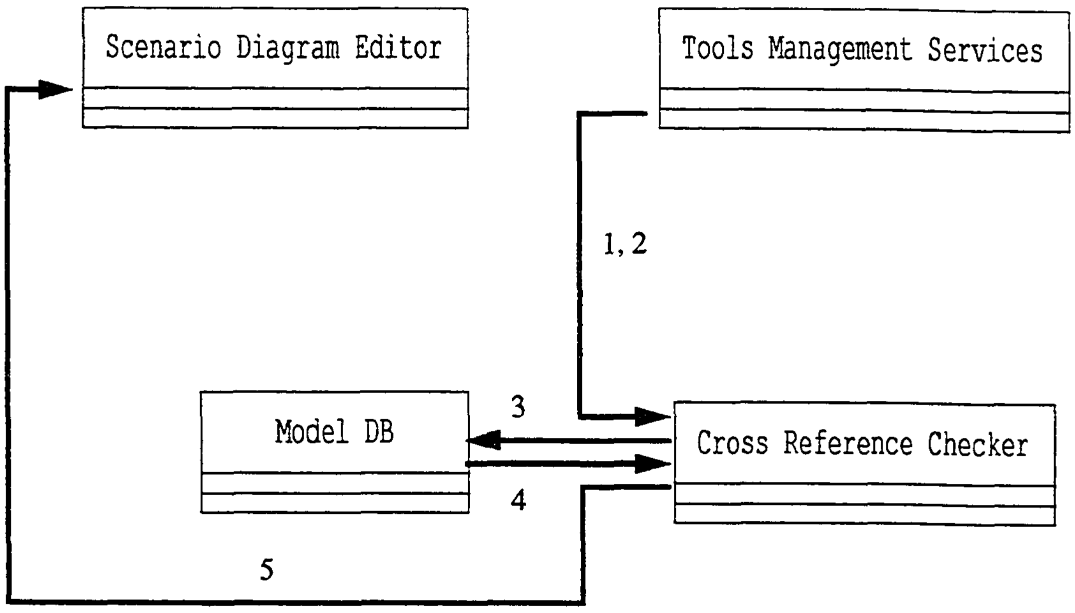
Scenario 1.1.1 : Level 2 : Multi-user access control

1. User ID and diagram ID of item of diagram model requested.
2. The information is passed to the access control services.
3. Access control services query model DB to see if diagram is locked.
4. Diagram is in use by another user. Access control services block.
5. User closes diagram.
6. Diagram ID is sent to SCM services.
7. SCM services update model DB and free the diagram.
8. OK sent to access control services.
9. Access control services sends OK to model DB.



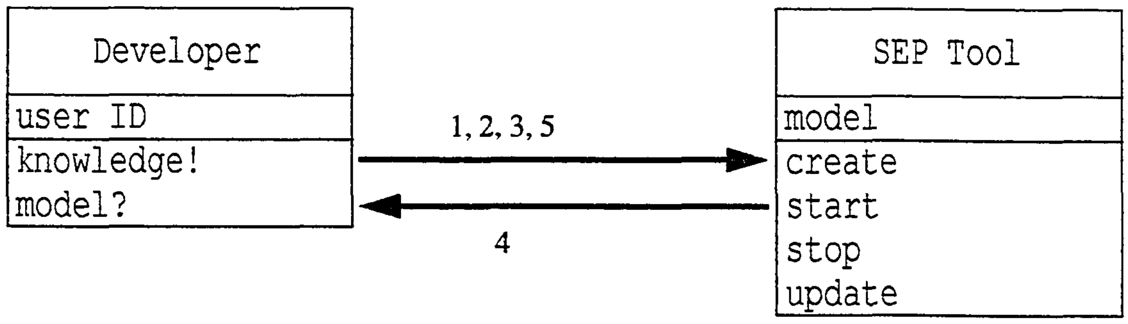
Scenario 1.2 : Level 1 : Model integrity is checked and diagram is stored

1. Developer adds information to the scenario diagram editor.
2. Developer requests the model to be updated.
3. Common user interface passes the information to the tools management services.
4. Tools management services invoke the semantic rule checker.
5. Tools management services invoke the cross reference checker.
6. Tools management services invoke the syntactic rule checker
7. Semantic rule checker returns OK to the editor.
8. Cross reference checker returns the subset of the diagram information that already exists in the model DB.
9. Syntactic rule checker returns OK to the editor.
10. The scenario diagram editor adds the diagram information excluding the part that already exists to the model DB.



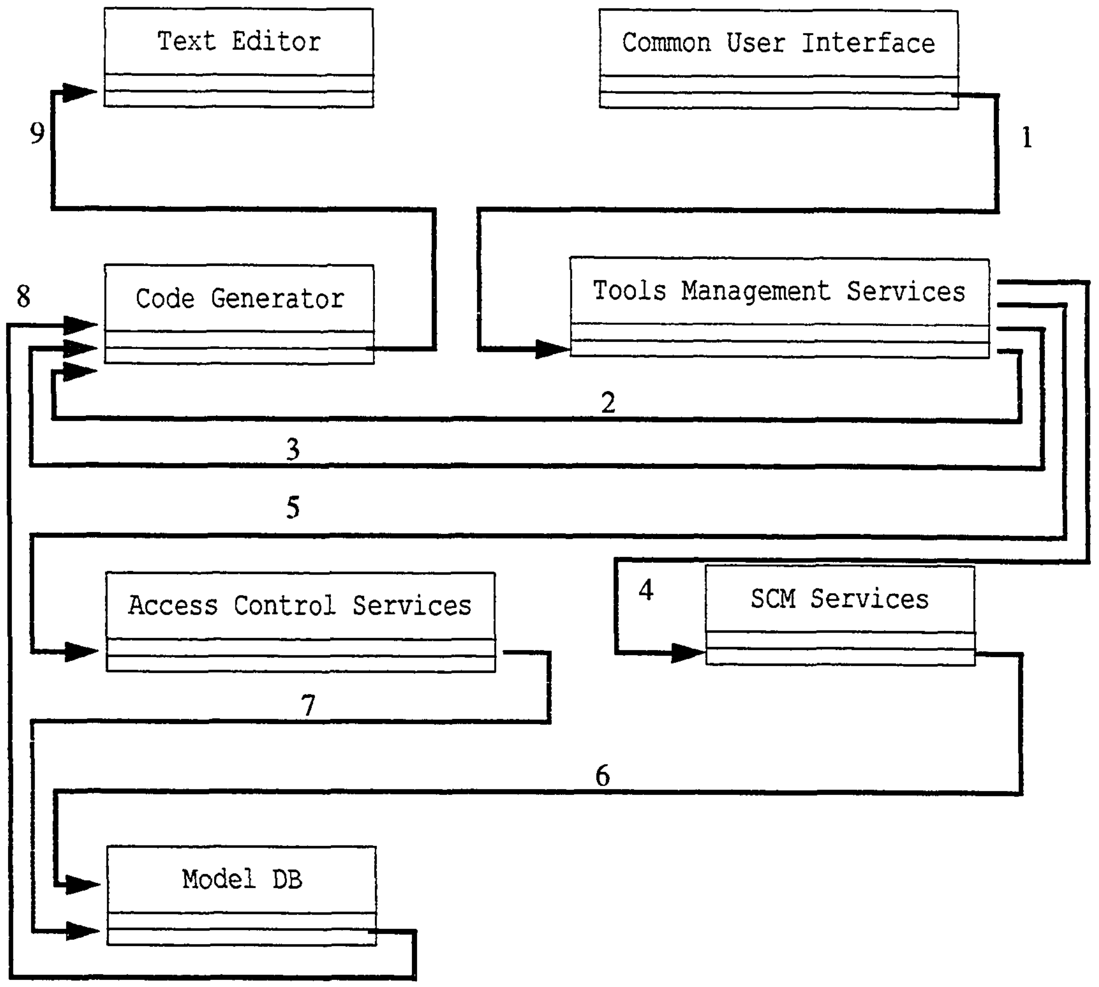
Scenario 1.2.1 : Level 2 : Cross reference check

1. Tools management services invoke the cross reference checker.
2. Tools management services send the diagram information to the cross reference checker.
3. Cross reference checker queries model DB.
4. Model DB returns components of the diagram that already exist.
5. Cross reference checker sends the query information to the scenario diagram editor.



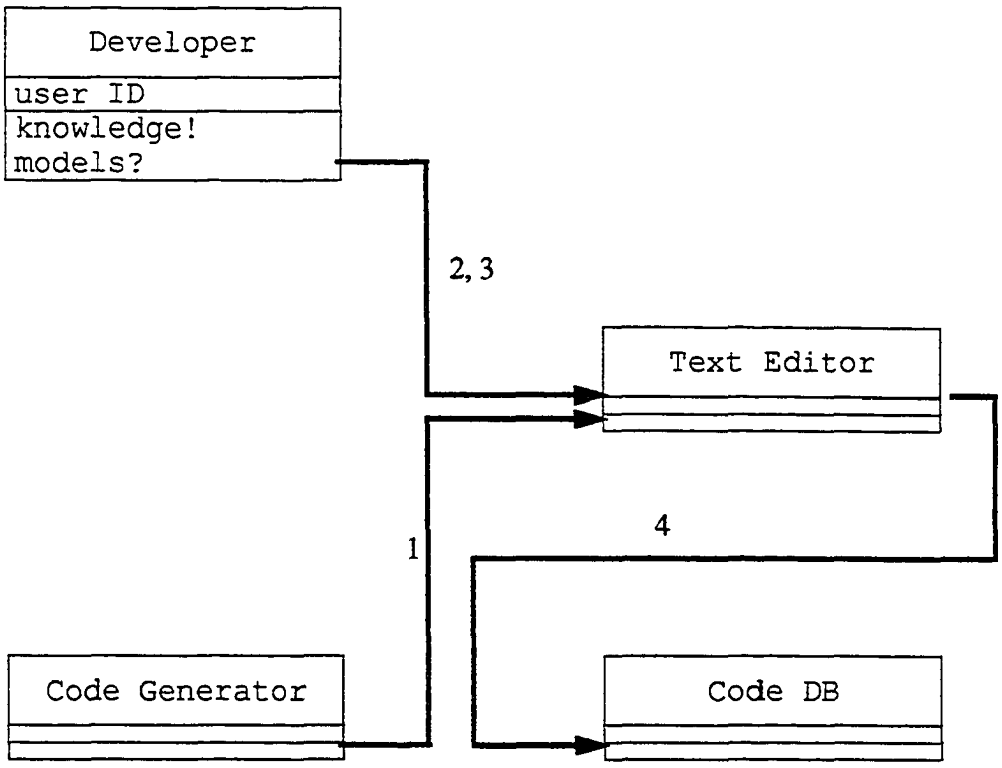
Scenario 2 : Level 0 : Developer generates code

1. Developer selects model
2. Developer selects object
3. Developer invokes code generator.
4. Tool generates code.
5. Developer modifies code (and possibly compiles and tests it) to implement object. Code is added to the repository.



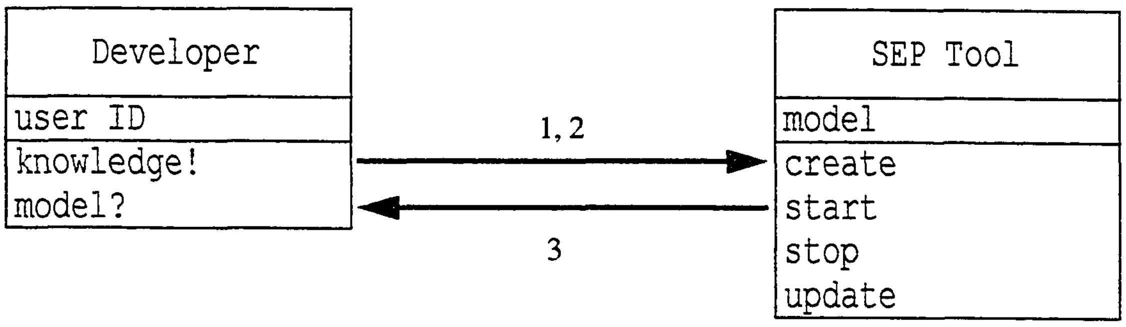
Scenario 2.1 : Level 1 : Code is generated

1. Common user Interface sends object selection and code generation request to tools management services.
2. Tools management services invoke code generator.
3. Tools management services invoke text editor.
4. Tools management services send selection information to SCM services.
5. Tools management services send selection information to access control services.
6. SCM services query model DB.
7. Access control services send OK to model DB.
8. Query results are sent to code generator.
9. Generated code is placed in text editor.



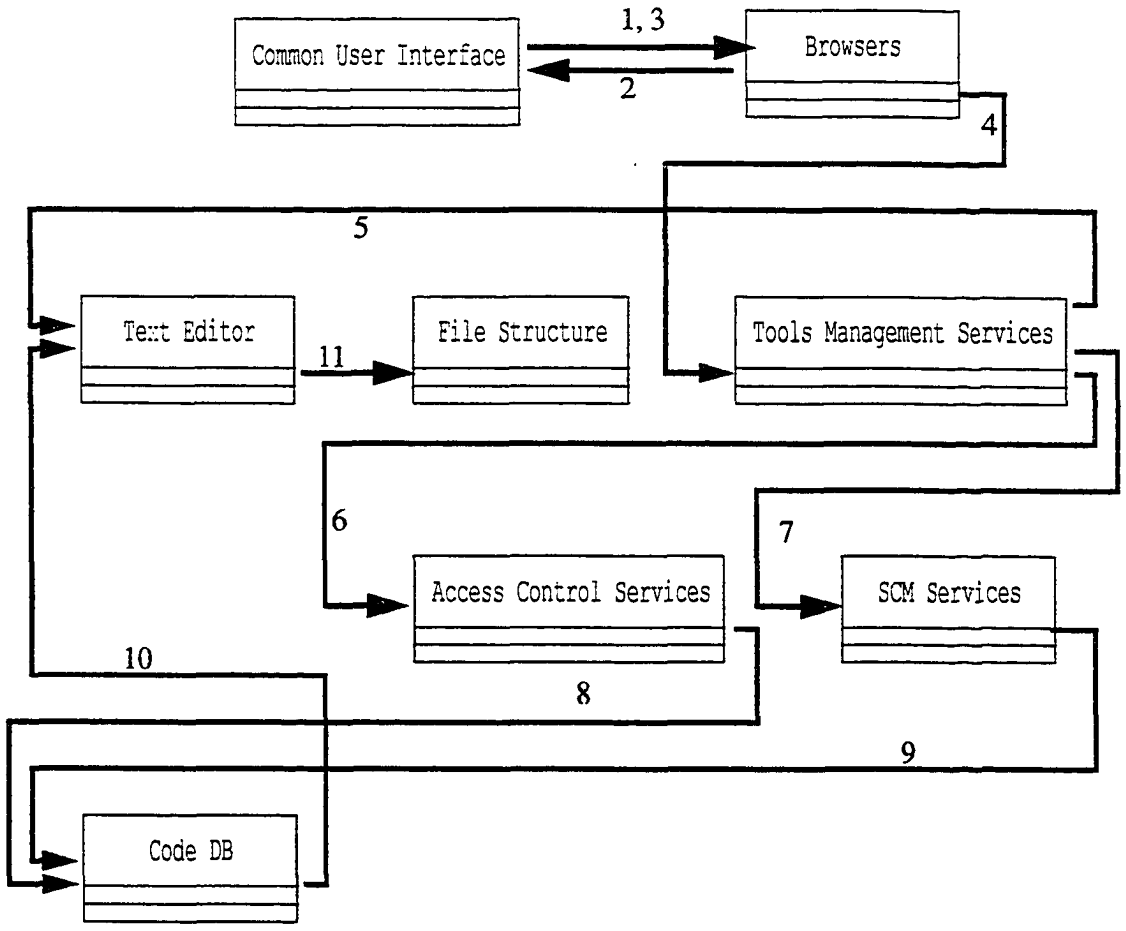
Scenario 2.2 : Level 1 : Code is modified and stored

1. Generated code is placed in the text editor.
2. Developer updates code.
3. Developer saves code.
4. Updated code is added to the code DB.



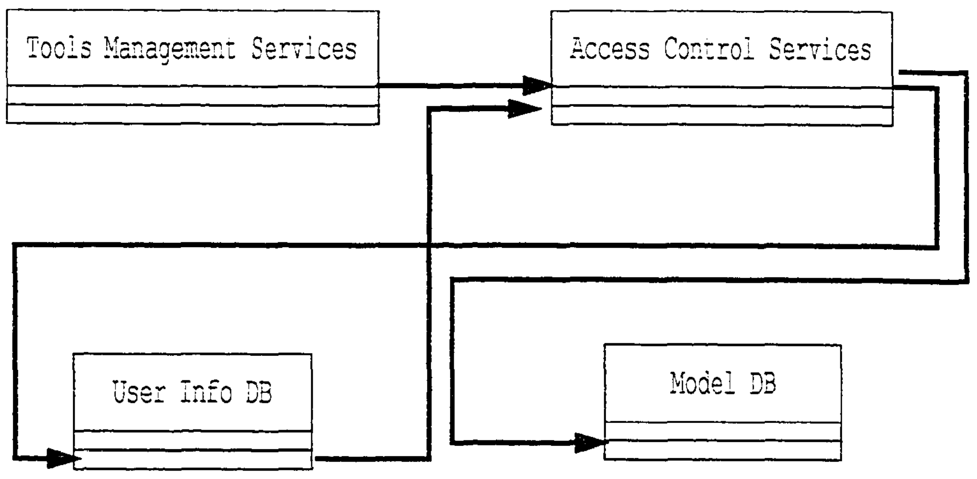
Scenario 3 : Level 0 : Developer reuses code

1. Developer browses model
2. Developer selects object
3. Tool provides code implementation of the object.



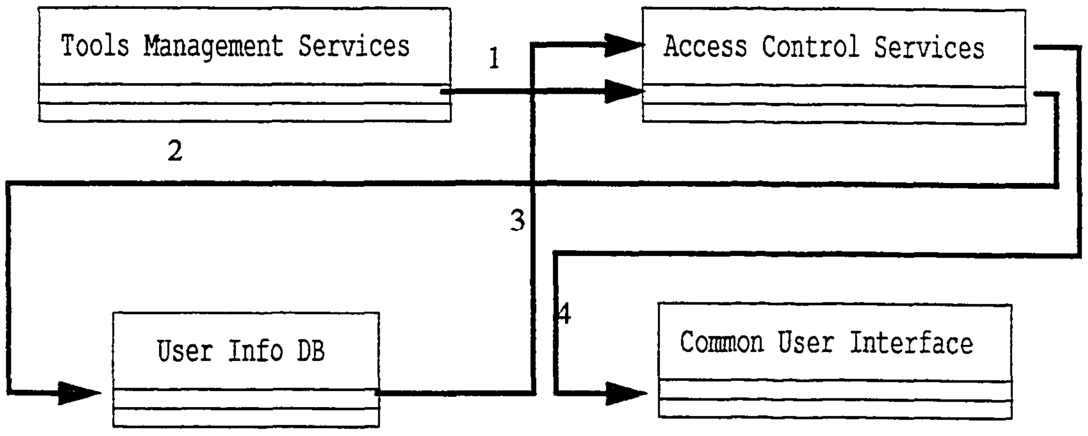
Scenario 3.1 : Level 1 : Developer selects and reuses code

1. Developer browses model through the common user interface.
2. Browser returns model information.
3. Developer selects object.
4. Browser notifies tools management services.
5. Tools management services invoke text editor.
6. Tools management services invoke access control services.
7. Tools management services invoke SCM services.
8. An OK is sent to the code DB.
9. A query is sent to the code DB.
10. Query result is sent to the text editor.
11. Code in text editor is saved in file structure.



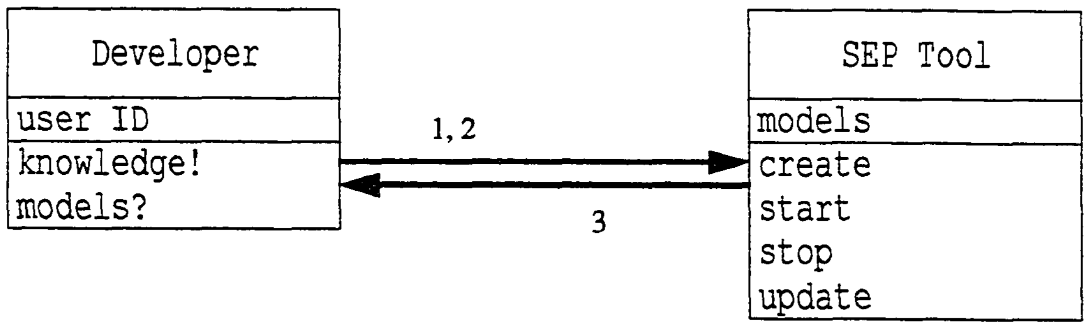
Scenario 3.1.1 : Level 2 : Developer has access authorization

1. Tools management services sends model and developer information to access control services.
2. Access control services queries user info DB.
3. Developer has reuse authorization.
4. OK sent to model DB.



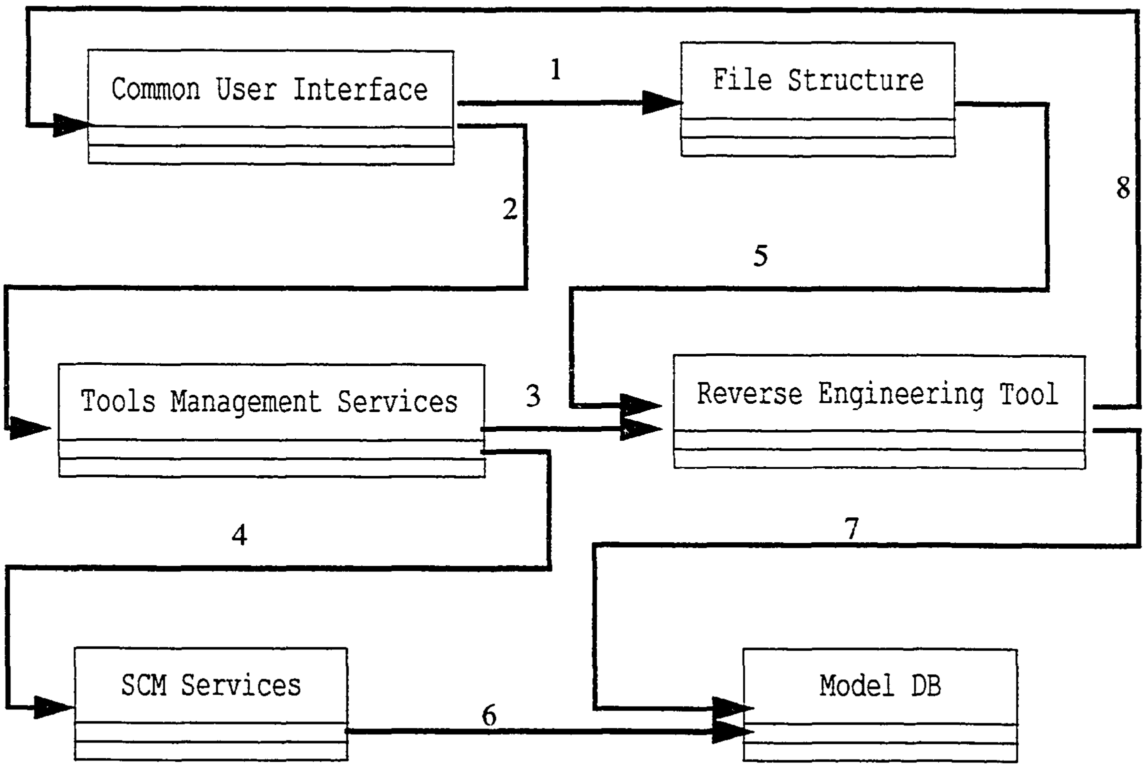
Scenario 3.1.2 : Level 2 : Developer is denied access

1. Tools management services sends model and developer information to access control services.
2. Access control services queries user info DB.
3. Developer has no reuse authorization.
4. Developer is notified through the common user interface.



Scenario 4 : Level 0 : Developer reverse engineers source code

1. Developer copies source code into file structure.
2. Developer selects reverse engineering option.
3. Tool reverse engineers code.



Scenario 4.1 : Level 1 : Source code is reverse engineered and stored

1. Developer copies source code into file structure using common user interface
2. Common user interface sends file information and model name to tools management services.
3. Tools management services invoke reverse engineering tool.
4. Tools management services sends model name to SCM services.
5. Reverse engineering tool accesses source code in file structure.
6. SCM services create model in model DB.
7. Reverse engineering tool adds reverse engineered model to model DB.
8. Reverse engineering tool sends "done" message to common user interface.

## REFERENCES

Bersoff, E.H., "Elements of Software Configuration Management," IEEE Transactions and Software Engineering, Vol.SE-10, No.1, January 1984.

Biggerstaff, T.J., and Richter, C., "Reusability Framework, Assessment, and Directions," IEEE Software, March 1987.

Boehm, B., "Software Engineering Economics," Prentice Hall, 1981.

Boehm, B., "A Spiral Model of Software Development and Enhancement," IEEE Computer, May 1988.

Bollinger, T.B., and Pfleeger, S.L., "The Economics of Reuse: Issues and Alternatives," Proceedings of the Eighth Annual Conference on Ada Technology, March 1990.

Booch, G., "Object Oriented Design with Applications," Benjamin/Cummings, 1991.

Buhr, R.J.A., "System Design with Ada," Prentice Hall, 1984.

"C++ Designer: User's Guide," Version 3.00, Meridian Software Systems, Inc., 1993.

Champeaux, D., and Farre, P., "A Comparative Study of Object Oriented Programming," Journal of Object Oriented Programming, April 1992.

Chen, M., and Norman,R.J., "A Framework for Integrated CASE," IEEE Software, March 1992.

Coad, P., and Yourdon, E., "Object-Oriented Analysis," Prentice Hall, 1991.

Cybulski, J.L. and Reed, K., "A Hypertext-Based Software Engineering Environment," IEEE Software, March 1992.

Dart, S.A., Ellison, R.J., Feiler, P.H., and Haberman, A.N., "Software Development Environments," IEEE Computer, November 1987.

Davis, A.M., Bersoff, E.H., and Comer, E.R., "A Strategy for Comparing Alternative Software Development Life Cycle Models," IEEE Transactions on Software Engineering, Vol. SE-14, No. 10, October 1988.

Fernstrom, C., Narfelt, K., and Ohlsson, L., "Software Factory Principles, Architectures, Experiments," IEEE Software, May 1992.

Giovanni, R.D., and Iachini, "HOOD and Z for the Development of complex Software Systems".

Goldstein, N., and Alger, J., "Developing Object-Oriented Software for the Macintosh: Analysis, Design, and Programming," Addison Wesley, 1992.

Haddock, G., "A Scenario-based Engineering Process: A Lifecycle Incorporating Formalisms, Tools, and Methodologies in Domain Specific Software Architecture," Master Thesis, University of Texas at Arlington, May 1993.

Harbison-Briggs, K., "A Scenario-based Engineering Process for the Semi-autonomous Surrogate Vehicle Program: An Approach for Task-Intensive Systems", Proceedings of the Tenth DARPA UGV Workshop, April 1993.

Hatley, D. J., and Pirbhai, I. A., "Strategies for Real-time System Specification," Dorset House, 1988.

HOOD Working Group, "HOOD Reference Manual," Issue 3.0, 1989.

HOOD Working Group, "HOOD User Manual," Issue 3.0, 1989.

Hufnagel, S., Harbison-Briggs, K., and Hammons, C., "Seamless Scenario-Driven Object-Oriented Approach: Methodology, Notation, and CASE Tool Integration," submitted for publication to ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, 1993.

Hufnagel, S., Harbison-Briggs, K., and Goldstein, D., "Scenario Driven Requirements Analysis Method," Proceedings of the Guidance and Control Software Initiative Workshop, December 1992.

Kemerer, C.F., "How the Learning Curve Affects CASE Tool Adoption," IEEE Software, May 1992.

Korson, T., and McGregor, J.D., "Understanding Object-Oriented: A Unifying Paradigm," Communications of the ACM, Vol. 33., No. 9, September 1990.

Mettala, E. and Graham, M. "The Domain-Specific Software Architecture Program," Crosstalk, Number 37, October 1992.

Meyer, B., "Object-Oriented Software Construction," Prentice Hall, 1988.

Mosley, V., "How to Assess Tools Efficiently and Quantitatively," IEEE Software, May 1992.

Murata, T., "Petri Nets: Properties, Analysis and Applications," Proceedings of the IEEE, Vol. 77 No. 4, April, 1989.

"OAATool: An Object-Oriented Analysis Tool for Windows," Version 1.2, Object International, Inc., 1991.

"Open SELECT CASE: Personal-SELECT Hood User's Guide," Meridian Software Systems, Inc., 1991.

Page-Jones, M., "Practical Guide to Structured Systems Design," Prentice Hall, 1988.

Pressman, R.S., "Software Engineering: A Practitioners Approach," 3 ed., McGraw Hill, 1992.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W., "Object-Oriented Modeling and Design," Prentice Hall, 1991.

Shlaer, S., and Mellor, S.J., "Object Lifecycles: Modeling the World in States," Prentice-Hall, 1992.

Shlaer, S., Mellor, S.J.,and Hywari, W., "OODLE: A Language Independent Notation for Object Oriented Design," Project Technology, 1990.

Spivey, J.M., "The Z Notation: A Reference Manual," 2ed., Prentice-Hall, 1992.

Sudkamp, T., "Languages and Machines," Addison-Wesley, 1988.

"Teamwork/Ada: User's Guide," Release 4.1, Cadre Technologies, Inc., 1991.

"Teamwork/HOOD: User's Guide," Release 4.1, Cadre Technologies, Inc., 1992.

"Teamwork/OOD: Object Oriented Design with Teamwork," Release 4.1, Cadre Technologies, Inc., 1992.

Thomas, I., Nejmeh, B.A., "Definitions of Tool Integration for Environments," IEEE Software, March 1992.

Wang, W., Hufnagel, S., Hsia, P., and Yang, S., "Scenarios Driven Requirements Analysis Method," Proceedings of the Second International Conference on Systems Integration, June 1992.

Wasserman, A.I., "A Graphical, Extensible Integrated Environment for Software Development," ACM SIGPlan Notices, January 1987.

Yourdon, E., "Modern Structured Analysis," Prentice-Hall, 1989.