# OOMMT

## Object Oriented Maintenance Management Tool

by

**BASSAM HADDAD**

Submitted in partial fulfilment of the requirements
For the degree of Master of Science

Project Advisor : Dr. NASHAT MANSOUR

Department of Computer Science
LEBANESE AMERICAN UNIVERSITY
March 2000

**LEBANESE AMERICAN UNIVERSITY**

**GRADUATE STUDIES**

We hereby approve the project of

**Bassam Haddad**

Candidate for the *Master of Science* degree.

**Dr. Nashat Mansour (Advisor)**

**Dr. May Abboud**

I grant to the LEBANESE AMERICAN UNIVERSITY the right to use this work, irrespective of any copyright, for the University's own purpose without cost to the University or to its students, agents and employees. I further agree that the University may reproduce and provide single copies of the work, in any format other than in or from microforms, to the public for the cost of reproduction.

# Table of Contents

# List of tables

# List of figures

# Acknowledgements

It is a great pleasure for me to acknowledge the assistance and help of few persons to this effort. My supervisor, Dr. Nashat Mansour, continuously encouraged, suggested various improvements and supported me to complete my project. I would like to acknowledge Dr. May Abboud who reviewed my work and provided me with valuable advises.

Finally, I gratefully acknowledge and thank all my family and friends for their patience, encouragement and contributions.

B.H.

# Abstract

OOMMT (Object Oriented Maintenance Management Tool) is a CASE tool that provides guidelines and procedures for carrying out a variety of activities performed during the maintenance process. This tool keeps track of change requests and reported problems specifically for object oriented systems. It helps the maintainer in controlling and managing the maintenance process through out its different stages. This tool is based on a 5 stages model that begins with the change request stage, passes through the change evaluation, maintenance specification, and maintenance design stages till it ends with the system release stage. At the end of each stage a form is generated. The model design corresponds to the traditional waterfall life-cycle model of software development. The tool's implementation is done in Visual Basic 5.0 as a front-end interface while all data are stored in Access 97 as a back-end database. Additionally a set of queries and reports brings ease and simplicity to the tool.

# Chapter 1 – Introduction

## 1.1 What is CASE Software ?

Computer-aided software engineering tools substantially reduce or eliminate many of the design and development problems inherent in medium to large software projects.

The ultimate goal of CASE technology is to separate design from implementation. Generally, the more detached the design process is from the actual code generation, the better. Many have recognised this basic software planning and structuring principle, and over the course of the last fifteen years, several *structured methodologies* have been developed and introduced to large numbers of programmers. These structured methodologies provided a design framework as well as a set of formalisms and practices, which have become the basis for software development. Although not perfect and largely relying on the thoroughness of the individual practitioner, these methodologies have allowed software developers to build more complex systems. Usually, these methodologies encourage the decomposition of large software systems into sets of small *modules*. The interfaces between these modules are well-designed by the software architect, allowing individual programmers to independently construct and test their respective assigned modules. Then during the final stages of the software development process, all of the modules are integrated to form the final program.

In many respects, CASE tools are a direct evolution of these early paper-based structured methodologies. Now many of those same structured methodologies and organisational techniques are being implemented as software programs themselves, instead of relying on the individual and engineers to religiously practice the methodologies (Fisher 1991).

One definition of computer-aided software engineering is the use *of tools that provide leverage at any point in the software development cycle.* This definition would include most tools that software developers are acquainted with today, including compilers, debuggers, performance profilers, and source code control systems.

A more restrictive but operationally better definition for CASE is the use of *tools that provide leverage in the software requirements analysis and design specification phases, as well as those tools which generate code automatically from the software design specification.* This more restrictive definition will be used in order to focus on the higher leverage design and code generation tools now being offered by CASE tool vendors.

CASE tools provide leverage by exploiting the design and development process, generally in the early stages, to yield implementation benefits later in the project. Requirements analysis and design specification packages are examples of CASE tools that allow the software designer to display and graphically edit a software schematic. Interactive graphics editing is certainly less time intensive, not to mention more enjoyable, than regenerating paper-based designs. Furthermore, these design

specification tools provide *consistency*-checking features, verifying that all software modules interface properly and all data structures are fully specified.

Other CASE tools are more tightly focused in objective than the design specification tools. An example of a focused tool is a *user interface* generator. Such tools include interactive form design and generation packages, which allow the software designer to develop and specify a form-based user interface for an application program. These tools usually operate in a what-you-see-is-what-you-get fashion.

## 1.2 Where can CASE help?

If the cause of most implementation shortfalls and reliability problems stem from improper or insufficient requirements analysis and design specification, what is the remedy? During the 1960s and 1970s, several structured methodologies were developed to impose rigid structures on the requirements analysis and design specification phases of the software development cycle. These methodologies presented a straightforward, disciplined approach to software development which, by following the methodologies, would greatly reduce the risks caused by requirements and design error.

Several of these methodologies (Fisher 1991) are Yourdon/DeMarco Structured Analysis *(data flow diagrams)* for requirements analysis, *Hierarchical plus Input and Output (HIPO)* charts for software module structuring, and *Warnier-Orr and entity-relationship diagrams for data modeling.* These methodologies all address different parts of the requirements analysis and design specification process. Newer methodologies include *object-oriented design and reengineering* (code refurbishment) (Fisher 1991). The knowledgeable software manager maintains these methodologies in his technological arsenal and knows when and how to employ them. He realises that several different methodologies will frequently be used on the same software project, each for its own speciality.

These formal, structured methodologies are the backbone of computer-aided software engineering. They provide the rigorous framework necessary to thoroughly specify and design software applications.

## 1.3 The Evolution of Software Tools

Software development tools are currently undergoing another revolution as CASE tools are forged to fortify the requirements analysis and design specification phases of the software development cycle. Today, all software developers are familiar with, and use, programming language compilers, symbolic debuggers, and commercially available subroutine packages. A large percentage of these professionals are acquainted with source code control systems for managing and organising large bodies of source code. And today, many are being introduced to computer-aided software engineering tools, that is, tools that help software developers specify the application's requirements and design.

Nowadays, the software industry is migrating from legions of programmers proficient in one programming language (usually COBOL, Ada, or C) and who participate in all phases of software development. The industry is favouring smaller groups of software designers and programmers, each highly skilled in one particular phase of the software development cycle.

This division of responsibility is common in large system application development, which has traditionally distinguished between *analysts, systems programmers, and application programmers.* Analysts are responsible for writing the requirements and design specification documents. Specialised database analysts develop database schema designs for new information management applications. Application programmers write the actual code based on the design specifications and database designs. Finally, test engineers integrate the modules and provide quality

assurance. This trend is one of specialisation. Other industries such as the construction industry, have undergone this process as they mature.

The advent of the personal computer elevated the need for better, more visual user interfaces as more and more white collar professionals began using computers. So now the software industry has a need for user interface specialists with the cognitive psychology skills necessary to design understandable man-machine interfaces.

Requirements analysis, also known as systems engineering, is becoming a profession in its own right. In the design phase, designers specialising in data modelling and structured design methodologies are responsible for transforming the requirements specification into a buildable software architecture. The structured analysis methodology models the underlying business function being implemented in software.

## 1.4 The Benefits of CASE Tools

For many software development organizations, the qualitative benefits of CASE tools outweigh the quantitative benefits. CASE tools employed in the early phases of software design and development yield lower costs and better results in the implementation and maintenance phases. This reduces the entire life-cycle cost. Design and development times will almost always be reduced by using CASE tools (Fisher 1991).

But perhaps their most satisfying benefit comes in the form of insurance, or peace of mind, that the job is being done properly, on schedule, and to the end-user's specification.

CASE tools yield a tremendous benefit in revealing many requirements (and surprises) before the implementation begins. (CASE tools are definitely not for adventure seekers!) However, much of the actual value received from computer-aided software engineering largely depends on how well it is integrated into the software development organisation.

**Complete Requirements Specifications.** Most software developers have witnessed the failure or rejection of a software application out in the field because what they built was not what the end-users wanted. Encouraging the software designer to completely specify the system's requirements in conjunction with end-users is the goal of requirements analysis and specification CASE tools. Most specification methodologies enforce end-user involvement because it is impossible to complete the specification without developing a model of the end-user's process or business function. Although the risk of creating an Edsel still exists (despite the best

efforts of all involved), the probability of doing so is greatly diminished with complete, detailed, and accurate requirements specifications.

**Accurate Design Specifications.** Design specifications are an integral component of many CASE tools. They encourage developers to specify thorough and complete architectures. There is nothing more frightening for a novice software engineer than facing the task of maintaining a large software system with incomplete, inaccurate, or non-existent design documentation.

In-line software documentation is rarely sufficient to communicate the system's architectural design without embroiling the reader in unnecessary detail. Often development teams say "we'll write the design specifications after the code has stabilised." Somehow, the documentation never gets written. A home purchaser contracting to build a custom new home would never let a contractor build the house first and then draw up the blueprint.

Furthermore, we have all seen designs that violate sound design practices: designs that expose unnecessary detail, encourage "spaghetti code,' and ignore the separation of concerns doctrine. (Separation of concerns is a software design principal that advocates organising software programs into groups of modules, each with well-defined inputs and outputs. Each module is treated as a black box whose intemal structure is not publicly known. Only the input and output parameters are visible to the other modules).

**Current Design Specifications.** Inaccurate design specifications not kept up-to-date relative to modifications made to the source code base can be an even bigger problem than incomplete design specifications. Many futile hours can be wasted

trying to understand a system's architecture from pouring over the design specification but not being able to reconcile it against the actual implementation.

CASE design tools can help maintain synchronization with the code implementation. Many tools actually attach the code to the specification, so that as the specification changes, so does the underlying code. Other CASE tools, most notably user interface design tools, automatically generate code. There is no need to maintain synchronization because you never touch the underlying code; only the design is edited.

**Reduced Development Time.** Completely specifying the software architecture substantially reduces, if not eliminates, waste from unnecessary or thrown-away code. Reducing such waste translates directly into reduced implementation time. Many software professionals feel most productive when they are sitting in front of a screen actually writing code. The constant gratification of a compile and run cycle is tremendously appealing to most of us.

Software developers must feel they are being as productive in the requirements analysis and design specification phases as they feel when writing code during the implementation phase. Because of the highly interactive, graphical orientation of most CASE tools, software developers derive an inherent satisfaction in designing elegant software architectures, similar to the thrill of actually writing code.

**Highly Extensible/Maintainable Code.** Any successful software project will never really be finished. End-users will either demand functional improvements or they will identify bugs in the software's operation, mandating some form of continuing development or maintenance work on the software. This becomes

particularly acute when the software application is an actual product for sale to end-users. It is easy to develop a design specification when embarking on a new software project or a major rewrite of an existing system; it is much more difficult to keep the design specification synchronized with evolutionary maintenance and enhancement work.

Although each minor enhancement or bug fix may not warrant an update to the design specification, the aggregation of several modifications will. It is difficult to enforce the discipline of periodic design specification updates under these conditions, especially when the maintenance has been left to a skeleton crew of one or two people.

It's difficult, even for CASE tools, to provide assistance in maintaining design specifications if the software built from those specifications is hand-written. But those CASE tools that automatically generate software from design specifications are not encumbered by this problem. In fact, this trend predominates in areas where software can be automatically generated, such as user interface design and database access design.

To summarise, CASE tools:

- Are more compelling to use than writing code.
- Help design rather than document.
- Maintain synchronization with the source code base.
- Reduce the risk of failure and surprise.
- Reduce total development time.

## 1.5 Scope of this project

This project is about developing a CASE tool that will guide, control and automate the maintenance management process of Object Oriented applications. As for all software, OO programs also need to be update and modified. Change Requests of any kind of software maintenance activities may be submitted. This Object Oriented Maintenance Management Tool (OOMMT) will help keep track of these CRs throughout the different stages of the maintenance process. Starting at the Change Request stage, stepping next into the Change Evaluation stage, the maintenance management spread out through many stages – where a form is generated as the output of each one – till it reach the System Release final stage. In addition, this tool will provide a set of queries and reports that will support the user in having a quick, efficient and easy access to various kind of information. This tool accommodates also different level of access permissions to users handling the process.

# Chapter 2 – Review of OO CASE Tools

## 2.1 A Taxonomy of CASE Tools

In this section we turn our attention away from software engineering practices and focus on computer-aided software engineering tools themselves. The technological underpinnings of computer-aided software engineering are nothing new. Most CASE tools implement pencil and paper structured design technologies developed during the 1960s and 1970s. These methodologies were popularized in many commercial data processing shops as a way to manage their application backlogs by reducing technical development risk.

During the 1980s, these analysis and design methodologies migrated into CASE tools as graphical workstations and personal computers became widely available. This migration is accelerating as the complexity of the design task grows beyond what can reasonably be accomplished using pencil and paper. Therefore, computer-aided software engineering technologies are evolutionary rather than revolutionary.

No single CASE tool or methodology can perform the entire specification and design job. Certainly several complementary methodologies are required to handle all facets of the software development job, from data structure design through user interface specification. Although there is a trend toward combining complementary methodologies into integrated tool environments, the universal tool still lives in the future (Fisher 1991).

**Database Fourth Generation Languages.** Fourth generation languages are high-level languages, which provide database access facilities. They are much easier to use than languages traditionally used for programmatic database access, such as COBOL and C. The goal is to remove the burden of tedious database access code by replacing it with a much smaller amount of code written in a higher-level 4GL specifically designed for database access or even with object oriented methodology. Many 4GLs provide form layout and design capabilities using common text editors and were the first technologies demonstrating the leverage of focused tools on coding from scratch.

With a 4GL, the application programmer can declare the input/ output screen forms presented to application program end-users. These forms are declared as sets of fields with well-defined properties, such as data input checking (e.g., integer range validation and legal data verification) and protected field display. The application programmer also specifies database access queries in the 4GL, allowing retrieved data to be displayed on the form and information input via the form to be added to the database.

**Data Modelling Tools.** In database applications, a single database is frequently shared by many different applications, each of which adds or extracts information. Before establishing a database for multiple applications and populating it with data, careful thought must be given to its content and architecture. Because database applications constitute a large proportion of commercial software applications, *data modelling* techniques were developed to help database designers architect a database to be both versatile and as efficient as possible.

Data modelling CASE tools help the database designer model this information flow throughout the firm and construct appropriate viewpoints for the various organizations requiring access. Data modelling tools help with the shallow semantics of databases. They make the semantics explicit so that everyone interprets the relationships among data items the same way. A well-designed, efficient database saves countless hours of application programming time as new applications are written to access the database.

**Analysis and Design Specification Tools.** Design specification tools generally use the *structured analysis and structured design* methodologies pioneered by Tom DeMarco and Edward Yourdon. These general-purpose specification and design tools can be used to specify and design almost any piece of software. *Analysis* and design tools usually implement data flow diagramming and structure charting techniques, and they are excellent for graphically depicting information flow between computational processes.

Many design specification tools have extensions for specifying temporal interactions prevalent in real-time control systems. Many tools also assist in composing documents required on defence-related mission critical software projects. Design specification tools are an excellent fit for the portions of an application where a focused tool doesn't make sense, such as internal calculation or kernel routines. The software designer must, of course, judge when a more focused tool, such as a user interface design tool, is called for and when a general-purpose design specification tool is appropriate.

**User Interface Prototyping Tools.** For many commercial applications, the user interface is the largest single component of the application program. User

interfaces vary greatly in style and content. Some, such as automated teller machines, are designed for ease-of-use by unsophisticated users, while other interfaces, like those in word processors, are built for high-volume processing.

Whatever the application, the user interface deserves special attention. We have all seen examples of user interfaces that have fallen short of their goal. Often, they differ only slightly from an interface regarded as highly successful. What differentiates a winning interface from a losing interface? Frequently, it is the time spent prototyping the interface and incorporating feedback from the end-user community that distinguishes the high-quality interface.

Prototyping can be a long and laborious process, and there is always the strong temptation to use the prototype as the final implementation, rather than redesigning based on end-user feedback. Fortunately, CASE tools for user interface design and generation are available in the commercial marketplace.

This type of rapid prototyping offers the leverage needed for highly productive and successful software projects. Often, just being able to display screen mock-up sequences is enough to open the communication channel between end-user and software designer. If the software designer has the capability to mock up interfaces rapidly (including non-functional ones), he has a vehicle for valuable feedback from the end-user community.

User interface CASE tools add value to the software development process during the requirements analysis and design specification stages. These tools leverage the designer by reducing implementation risk and greatly enhancing end-user acceptance.

**Code Generation Tools.** *Automatic code generation* is the ultimate goal of most CASE tool vendors and certainly of all CASE tool users. Code generation is the

ability to automatically generate compilable software directly from a design specification. Ultimately, the software designer's time is much better invested specifying and designing rather than coding and debugging. Unfortunately, true code generation is not available in any of today's general-purpose tools or so-called application generator products. But code generation is available in a variety of focused tools, especially in user interface design tools.

In the past, CASE technology focused on general-purpose requirements analysis and design specification. New developments in CASE technology now emphasize specialized development tools. Special development tools focus on one particular type of software, such as database access and user interface development. For example, there are a growing number of forms generation packages in the marketplace, ranging in sophistication, price, and delivery environment (PC to mainframe).

Future tools will attack the more general problem of automatic code generation. Automatic programming is a difficult problem and is still largely considered a research topic. Still, each new tool makes small innovations in this area, and eventually, code generation will be commonplace.

## 2.2 Characteristics of Good CASE Tools

A simple taxonomy of CASE tools might pigeonhole tools by the software development phase where they add leverage, such as requirements analysis, design specification, or implementation. However, most of the general-purpose tools span several development phases, usually the requirements analysis and design specification phases. These tools typically implement the structured analysis (Yourdon/DeMarco) methodology, and are able to transform data flow diagrams and mini-specifications into structured designs including data structure definitions and module hierarchies.

As with all developing technologies, certain aspects of CASE are more advanced than others. This makes it difficult to summarise today's state of the art. However, there is a general trend emerging of tools being built to cover the entire software development cycle, including automatic code generation and maintenance control systems.

Many of today's vendors are recognizing that their tools cannot focus on just one or two facets of the development process, such as analysis and design specification or user interface layout. Rather, tomorrow's successful tools must deal with all phases of the development cycle and tackle the fundamentally difficult problem of automatically generating error-free software directly from design specifications. This code generation barrier is beginning to crumble with innovations being made in many of today's CASE tools and will further diminish during the next decade as they advance CASE technology, expand the scope of their tools, and develop standards.

Fundamentally, CASE tools must meet several criteria in order to be successfully adopted as part of a software developer's tool kit. Meeting these criteria is essential to the tool's acceptance into the development organization's standard practices (Fisher 1991).

CASE tools must:

- **Simplify.** A major goal of CASE technology is to decompose requirements and designs into manageable components. Their function is to simplify, explain, and reduce.

- **Serve several audiences.** CASE tools for the requirements and design phases of the software cycle serve several masters. Their output must be understandable by the end-users and the organization sponsoring the software development. In addition, the tool must provide real design value to the developers themselves.

- **Save time and money.** Using a CASE tool should be cheaper and more efficient in the long run than building the software system using traditional methods. CASE tools should substantially reduce implementation and maintenance efforts by yielding higher-quality specifications and designs.

- **Produce quantitative and verifiable designs.** The specifications and designs generated by CASE tools must accurately and concisely articulate the software features and components to be built. Each requirement in the software implementation must be verifiable and traceable back to the requirements

document. Performance criteria, boundaries, and error conditions must be incorporated as part of the design.

- **Support change.** Specifications and designs produced using a CASE tool must be adaptable as the requirements and design goals of the project change. A design document that falls out of synchronization with the underlying code becomes useless and may cause developers to waste time in future enhancements to the software.

- **Show, not say.** Good CASE tools present specification and design information visually. CASE tools are to software engineering what CAD (computer-aided design) programs are to schematic design and layout. For end-users and developers alike, it is much easier to comprehend a graphic illustration than to read several pages of text description.

## 2.3 Emerging CASE Tools

Object-oriented design is an emerging CASE methodology designed to support the newly prominent object-oriented languages, particularly C++. C++ is a strong successor to the C language popular in microcomputer and minicomputer applications, and is rapidly being adopted in many minicomputer and microcomputer application development projects. Other languages, like Ada, also are suited to object-oriented design.

Object-oriented languages are unique because they inherently modularise data structures and compartmentalise code. Object-oriented languages attempt to make good structure inherent in the language rather than relying on the completeness of the designer. This provides a built-in mechanism for achieving modularity, reusability, and generality; goals that more traditional programming languages achieve only through careful analysis and design before implementation begins.

## 2.4 Object-Oriented CASE Tools

Sometimes, people have been seduced into thinking they are going to get more out of a CASE tool than is possible. In attempting the impossible, they produce disastrous results. The original idea behind CASE tools was to spare systems analysts and software designers from spending their valuable time in drafting activities by providing them with software that automates the production of graphically oriented analysis and design methodology deliverables. If solving that critical problem is kept in sight, and the continuing need to have highly skilled, well-trained analysts and designers is recognized, use of CASE tools can produce significant productivity improvements.

### 2.4.1 The Role Of Case Tools In Object-Oriented Rapid Prototyping

Any software that supports the software engineering process in any way can technically be considered a CASE tool. This can include debuggers, test support, language-sensitive editors, and many other programming-oriented tools. These are often referred to as "lower CASE" tools because they are used toward the end of the lifecycle, and the tools that support analysis and design are sometimes called "upper CASE". For rapid prototyping, we are primarily interested in the upper CASE tools.

For upper CASE tools, one of the big selling points is the ability to check the developer's specifications against the rules of the methodology in use. This is called *consistency checking.* Object Oriented analysis and design methods have fewer rules

than structured analysis and design (SA/SD). Simplification is primarily due to the absence of elaborate functional and data hierarchies and decomposition, and to the absence of format switching between analysis (data flow diagrams) and design (structure charts). Using SA/SD, one moves from data flow diagrams, control flow diagrams, and entity relationship diagrams into architecture diagrams and finally to structure charts through transformation and transaction analysis. Hierarchical decomposition causes many problems with partitioning, balancing, leveling, and conservation; there are complicated structured rules for these elements of a structured specification. Without hierarchical decomposition, there is much less need for consistency checking.

Object-oriented CASE tools need to do less and should therefore cost less than structured CASE tools. The object-oriented products are primarily drawing tools, explaining much of the decrease in price. Drawing tools that are just drawing tools with no built-in object-oriented constructs cost even less. Drawing is at least 80 percent of what most users do with a CASE tool. For some CASE tools, however, drawing is their weakest feature. This leads to the conclusion that selection of simple drawing software instead of a CASE tool to support OOA and OOD is often the right decision! (Connell and Shafer 1995)

In the area of prototyping, some high-end CASE products have the ability to generate a software "prototype." In some instances, the prototype generated is of fairly impressive quality. When this happens, there is a positive and strong coupling between the software and the specification. Warning to the reader: Don't be seduced! A well-constructed drawing can appear to be a good design when, in fact, it may bear little relationship to the real user requirements and may actually represent a poor design. The "if it was produced on a computer, it must be right" syndrome applies

also to CASE tools. Attractive drawings make impressive presentation materials, but may or may not be adequate as an application design.

A prototype that is easy to iterate cannot be generated by a CASE tool unless there is a means of debugging at the drawing stage. To put it another way, when the requirements commissioners see the prototype and want changes, what will be changed-the software or the drawing? If the software is changed, it will quickly lose all coupling with the drawing. Is the prototype generated in a language that is easy to modify (many products generate C or C++)? If changes will always occur in the drawing, there should be consideration given to whether or not the drawing tools provide an environment suitable for debugging thorny software problems. At present, we know of no CASE tool vendors who recognize this problem, much less provide a good solution. They are often too busy selling code generators.

When requirements commissioners' request changes to the prototype, the temptation is to skip the object-oriented CASE tool and go directly to the prototype software, which is easy to access and change. Developers find instant gratification in viewing the results.

It often appears that placing the CASE models between the requested changes and the prototype is time-consuming and unnecessary. This is a dangerous situation because it will tempt prototypers to skimp on the models and only change the prototype after obtaining feedback from requirements commissioners. Iterations that skip updates to the models and go directly to the prototype can appear to be quicker at first, but the short-term productivity gains cannot last, since complex changes will be much more difficult without a good overall system roadmap. When the prototype gets too far ahead of the models, trying to reverse-engineer the specs out of the prototype becomes an arduous and sometimes impossible task.

A good OOA/OOD CASE tool has enough intelligence about the methodology being used (that does much of the work for the specifier) and supports changes in a methodology-smart way. Graphic objects that should be connected stay connected. References to attributes in service specs might be automatically updated when the attributes are modified. The service specs, attribute definitions, and graphic models are all easy to transfer to a word processing document when it is time to create a deliverable specification. Features like these will make even the most senior developer appreciate the value of using such a tool concurrently while prototyping -as long as it doesn't cost too much and is a good drawing tool as well (Connell and Shafer 1995).

### 2.4.2 Case Tool Shortcomings

An understanding of common weaknesses found in many CASE products will perhaps be helpful for comparing products prior to an acquisition. The most important weakness to understand is the drawing deficiencies. Often there is a significant delay (several seconds) between the time the user commands a change to the drawing and the time the change appears on the screen. This is most often due to the fact that a multi-user repository is being updated. The changes are being written to a complex data management system. This excuse, although understandable, is not acceptable. A computer-aided drawing tool must be at least as fast to use as pencil and paper.

Another factor at work in the selling (or overselling) of object oriented CASE is market economics. The market for personal computer drawing tools is vastly larger

than the market for CASE tools. Thus, comparatively large efforts have gone into understanding what users want, in terms of the look, feel, and operation of the PC based drawing tools. If CASE vendors were smart, they would extend this look and feel into their products, but they often don't.

A common weakness of CASE products is their lack of adequate report-generation capability (Connell and Shafer 1995). Again, the CASE vendors could benefit greatly by studying how related PC-based products, such as database management systems and many project management systems, provide this functionality. Virtually every popular PC database product has an easy-to-use, flexible report writer that allows the user to create reports selected and formatted in any manner desired without having to do any programming. Since a CASE tool is essentially a specialized type of data repository, why aren't there better CASE report-writer modules that would allow analysts and designers to create customized requirements and design specifications according to whatever tailored version of whatever documentation format they are using on their project? Instead, we get inflexible templates for standard formats and graphics that are, for some strange inexplicable reason, difficult to transform into figures in the specification. It seems that the primary objective of analysis and design - to produce a specification document – is forgotten.

Another common weakness of CASE tools is their inability to keep up with changes in the methodology they support (Connell and Shafer 1995). Developers of methodologies are continually making refinements to the guidelines and notation. This requires the vendor of a CASE tool that supports the modified methodology to make changes to their software. Many of the more popular methodologies are modified every six to twelve months and most software developers have difficulty

coming up with new versions of their products that frequently on a sustained basis. This is why we saw tools that mainly supported flow charts when dataflow diagrams were becoming the most prevalent approach to analysis. When the industry had mostly accepted the DeMarco approach to structured analysis, the CASE tools were mostly supporting Gane and Sarsen structured analysis. By the time vendors had switched to DeMarco, analysts wanted to do Ward/ Mellor real-time structured analysis. Many of the leading Case tool products were becoming old by software standards (over five years) by the time they offered good support for Ward/Mellor. This meant that they were probably becoming extremely difficult to modified (most were probably written using a hierarchical, procedural approach with a third-generation language). Therefore, when analysis changed again, first to information engineering and then to object oriented, the old software, in most instances, failed to keep up. This made room for a whole new crop of products that had never been anything but object-oriented. But, as soon as those products became available, the rules and notational conventions for object-oriented analysis changed. This cycle seems to be immutable and one of the tragic flaws of CASE tools.

One final word about keeping up with the methodology. A relatively recent strategy that attempts to solve this problem is the development of CASE tools by the methodologist for the methodology they support. The idea is that the methodologist will have the next version of the CASE tool ready to ship concurrently with the next version of the tool. Also, good methodologists are not necessarily the best software developers. In the case of one methodologist, the drawing features of his company's CASE tool suffer a bit in comparison to his fine approach to object-oriented analysis and design (Connell and Shafer 1995).

The best bet is to keep the decisions of which methodology to use separate from which CASE tool to use. Let applications (systems under construction) drive decisions about what methods and techniques to select for the job, and in turn, let the methods and techniques drive decisions about what tools to purchase. So often, that important order of events is reversed. Selecting a tool first and then deciding what applications can be built with it is limiting (when you have a hammer, every problem looks like a nail). Always pick a methodology that is described in a book that can be obtained in any good technical library. Then pick the best CASE tool that supports that methodology.

Another weakness of CASE tools is the consistency-checking that justifies their use over computer-aided drawing tools. There seem to be tools of just two types with respect to consistency checking. The first type does almost nothing in terms of consistency checking and is, therefore, little more than an expensive drawing tool. The other type does an overwhelming abundance of consistency checking. The smallest set of models will cause the latter type of tool to generate many boring pages of consistency checks, mostly trivial and sometimes even erroneous. Of course, most tools allow consistency-checking to be optional. The problem is that searching through the trivial and wrong errors for the important errors can be so tedious that it often makes the whole feature of automated consistency checking virtually worthless. It is usually more efficient to subject models to peer review, providing at least some element of reasonableness. CASE software with overblown consistency checking is little more than an expensive drawing tool.

Finally, there is the universal absence of strong coupling between the models and the software application developed from the models. With all this automation and the high price tags, doesn't it seem like there should be some way of guaranteeing, or at

least checking, that the application does what the models claim it does? Actually, tools may be too difficult for vendors to create in the form of an affordable product. Vendors have been taking the approach of code generation, but we have already pointed out the fallacy of this approach, It may be a nice feature to use once in a while, but it will not guarantee consistency between code and model. We know of one vendor who has a product that can digest C++ and Smalltalk and produce Coad/Yourdon diagrams, but it is strictly a reverse engineering tool and has little in the way of support for analysis and design. Perhaps a tool such as this could be used in combination with an OOA/OOD CASE tool, but that could be awkward. Would convergence of the two models ever be achieved? No tool that we know of simply looks at the source code and produces a listing of consistency errors in comparison with the models. What would be wrong with that approach, if it was accurate, reasonable, and not overblown?

We often advocate the use of CASE tools to our clients while countional conventions to represent the components (objects, processes, data stores, dataflows, messages, instance connections, etc.) of the methodology. Internal to vendor software is a mapping between the shape the user draws and the methodology component the shape represents. Capability should be provided that allows this mapping to be modified by the user. If, for instance, an object is represented by a box with rounded corners, the user should be able to command that henceforth, objects are represented by square boxes or circles or clouds or whatever shape is desired. Then the tool would know to generate the new shape whenever the user wants a new object. This modifiable shape-mapping feature would go a long way toward allowing users to

customize the tool in order to help it keep up with methodology changes (Connell and Shafer 1995).

The problem with generating custom reports could easily be fixed with the standard data export features found in most good personal computer packages. Drawings, data dictionaries, method, module, or service specifications should all be exportable in formats usable by other desktop software. Then the user could pick up the data with a favourite database, word processor, spreadsheet, or desktop publishing package, format the document, and print out whatever type of document is required. This type of export feature would be effort much better spent than trying to develop the most excellent report writer or the most wonderful set of standard specification templates. There is no such thing as standard specification formats, because software developers always tailor the standard format they are using for each new project.

An expert system for allowing power users to add their own consistency-checking rules and to specify which rules are to be checked against which models would be extremely valuable.

Tools to generate code that is tightly coupled to analysis/design models will be expensive to develop. What is needed is an executable, interpreted, high-level scripting language that the analyst can use to write service specifications from within the CASE tool coupled somehow with an object-oriented GUI builder, such as XVT. There would also have to be good debugging tools for the scripting language. Then when prototype and model are approved, at the end of prototype iteration, an automatic translator should be available to translate the service specification scripts into C++ or Smalltalk, automatically encapsulated with their specified attributes in their specified objects.

## 2.5 CASE practices

Hypothesis 1. There are common problems faced by organisations in using CASE tools.

Hypothesis 2. Organisations have realised benefits with the use of CASE tools.

Operational systems which support day-to-day running of the organisation formed the bulk of software systems developed by the organisations, followed next by decision support systems which facilitate management decision making, then inter-organisational systems which provide links to business partners and finally, other types of systems. Developing new applications was the main activity of the IS(Information Systems) departments, followed by maintenance of applications as the second most common activity while end user computing support together with technical operations support tied for third place in the ranking of activities carried out.

The majority (53.7%) of IS departments functions as independent units reporting directly to the Chief Executive Officer of the organisation while 33.3% of the IS departments were subsumed under another functional unit in the organisation (Poo and Chung, 1998).

### 2.5.1 Current practices on the use of CASE tools

Table 1 shows that only 29.6% of organisations which responded used CASE tools although 68.5% of these organisations used a formal software development methodology.

The majority (43.8%) of organisations, which had CASE tools, used them in a quarter or less of their software systems under development (see Table 2). 31.3% of them used the CASE tool for between a quarter to half of their systems under development. Only 18.8% of them used their CASE tool for more than 75% of their applications under development.

According to Table 3, the usage of CASE tools for maintenance of software systems was even lower. 62.5% of organisations with CASE tool used it to maintain up to only a quarter of their systems under maintenance. Only 12.5% of those with CASE tools used it to maintain more than 75% of their systems under maintenance.

In a case study conducted by Mary Sumner of Southern Illinois University (Bergin 1993) on 13 project managers who used CASE tools, it was found that approximately 40% of new applications were developed with CASE tool support while only 21% of maintenance projects had CASE tool support. Hence, the observed trend is that CASE tools are used more for software development than software maintenance.

Table 4 shows that the four most common activities for which CASE tools were used.

· Documentation (62.5%).

· System design (56.3%).

· Requirements analysis (50%).

· Drawing diagrams (50%).

The other more common activities which involve the use of CASE tools were information systems planning (37.5%), coding or code generation (37.5%), prototyping (31.3%), project management (25%) and change management (25%).

Table 1 - Organisation using CASE tools (Poo and Chung, 1998)

| Response | Freq. | Percent(%) |
|---|---|---|
| Yes | 16 | 29.6 |
| No | 36 | 66.7 |
| (Missing) | 2 | 3.7 |
| Total | 54 | 100.0 |

Table 2 - Development using CASE tools (Poo and Chung, 1998)

| Response (%) | Freq. | Percent(%) |
|---|---|---|
| 0-25 | 7 | 43.8 |
| 26-50 | 5 | 31.3 |
| 51-75 | 1 | 6.3 |
| 76-100 | 3 | 18.8 |
| Total | 16 | 100.0 |

Table 3 - Software maintained using CASE tools (Poo and Chung, 1998)

| Response(%) | Freq. | Percent(%) |
|---|---|---|
| 0-25 | 10 | 62.5 |
| 26-50 | 3 | 18.8 |
| 51-75 | 1 | 6.3 |
| 76-100 | 2 | 12.5 |
| Total | 16 | 100.0 |

Table 4 - Activities for which CASE tools are used (Poo and Chung, 1998)

| Rank | Response | Freq. | Percent(%) |
|---|---|---|---|
| 1 | Documentation | 10 | 62.5 |
| 2 | System Design | 9 | 56.3 |
| 3 | Requirements Analysis | 8 | 50.0 |
| 3 | Drawing Diagrams | 8 | 50.0 |
| 4 | **Information Systems** Planning | 6 | 37.5 |
| 4 | Coding | 6 | 37.5 |
| 5 | Prototyping | 5 | 31.3 |
| 6 | Project Management | 4 | 25.0 |
| 6 | Change Management | 4 | 25.0 |
| 7 | Testing | 3 | 18.9 |
| 8 | Configuration Management | 2 | 12.5 |
| 9 | Others | 1 | 6.3 |

The remaining activities which involved the use of CASE tools were testing (18.9%), configuration management (12.5%) and others (6.3%).

In Sumner's study the three activities for which CASE tools were most frequently used were system analysis, system design and documentation. It was also observed that CASE tools were used to support isolated activities within the system development life cycle, rather than being integrated throughout the life cycle.

### 2.5.2 Perceptions of benefits and barriers on the use of CASE tools

Many claims have been made about the benefits of CASE. One of the major benefits of CASE is the introduction of engineering-like discipline into the software process. Using CASE tools, a software engineer can take advantage of diagramming tools, design checking tools and a disciplined methodology (Poo and Chung, 1998).

As shown in Table 5, the four most significant benefits realized with the use of CASE tools were as follows.

- Facilitates drawing of diagrams (62.5%).
- Improves software maintenance (56.3%).
- Creates a repository for system documentation (56.3%).
- Provides checks on analysis and design errors (56.3%).

The other more significant benefits perceived, each with a response of 37.5%, were as follows.

- Increases user involvement in system design.
- Creates an enterprise-wide data dictionary.
- Aids in project management and control.

The less significant benefits from using CASE tools are perceived to be:

- Support for software engineering methods (31.3%).
- Ensures conformance to system development and maintenance standards (31.3%).
- Supports prototyping (25%).
- Others (6.3%).

Table 5 - Benefits realized with use of CASE tools (Poo and Chung, 1998)

| Rank | Response | Freq. | Perc.(%) |
|---|---|---|---|
| 1 | Facilitates drawing of diagrams | 10 | 62.5 |
| 2 | Improves software maintenance | 9 | 56.3 |
| 2 | Creates repository for system documentation | 9 | 56.3 |
| 2 | Provides checks on analysis and design errors | 9 | 56.3 |
| 3 | Increases user involvement in system design | 6 | 37.5 |
| 3 | Creates enterprise-wide data dictionary | 6 | 37.5 |
| 3 | Aids project management and control | 6 | 37.5 |
| 4 | Support for software engineering methods | 5 | 31.3 |
| 4 | Ensures conformance to system development and maintenance standards | 5 | 31.3 |
| 5 | Supports prototyping | 4 | 25.0 |
| 6 | Others | 1 | 6.3 |

In the study conducted by Sumner, the five most significant benefits of using CASE tools, were as follows.

· Vehicle for using structured design.

· Prevents re-drawing of diagrams.

· Provides improved maintenance.

· Increases user involvement in system design.

· Creates a repository for design documentation.

This shows that CASE tools are perceived to be quite useful as a diagramming tool, a repository for system documentation and a means to improve software maintenance.

The following factors were considered as barriers to the use of CASE tools:

· High cost of implementing CASE tools.

· Long learning curve to use CASE tools effectively.

· Limited capability of CASE tools.

· Lack of fit between system development methodology and CASE tools.

· Using CASE tools without knowledge of underlying software engineering methods and techniques.

· Uncertainty over the benefits of CASE tools.

In the study conducted by Sumner, both CASE and non-CASE users felt that the limited capability of CASE tools was the most significant barrier to the use of CASE tools. For CASE users, other significant barriers were the long learning curve and lack of fit with current methodology. For non-CASE users, the other very significant barrier was the high cost, which, not surprisingly, was the least of the

CASE users' concerns. Resistance by system developers was ranked last and second last by non-CASE and CASE users, respectively.

A series of tests were conducted to find out whether there were significant differences in the practices between organisations with different demographic characteristics (Poo and Chung, 1998).

***Number of IT*** professionals in organisation: It was found that organisations with 20 or more IT professionals had a higher tendency to adopt a formal system development methodology and use CASE tools than organisations with less than 20 IT professionals. This supports the assumption that larger IT departments would take the lead in adopting software engineering methods and tools since a larger IT department is an indication of the size and complexity of projects undertaken and the importance of the IT function to the organisation.

***Number of employees*** in organisation and industry type: Intuitively, the size and nature (industry type) of the organisation would influence the number of IT employees in the organisation and its software engineering practices. However, a higher percentage of smaller organisations (i.e. those with less than 500 employees) made use of a formal system development methodology and CASE tools compared with larger organisations(31.0% vs. 28.0%). In addition, there is no significant difference between the various types of industry in terms of number of IT professionals employed, the use of a formal system development methodology and the use of CASE tools.

***Industry sector***: Interestingly, the results from the survey show that although the public sector led in the usage of formal system development methodology (78.6% vs. 65.0%), the private sector led in the usage of CASE tools (30.0% vs. 28.6%). With regards to the use of CASE tools, the public sector may be lagging slightly behind the private sector, because public sector organisations usually need to justify the purchase of CASE tools with concrete benefits. In the private sector, the purchase of CASE tools may not be governed so strictly by the results of a cost-benefit analysis, thus leading to more wide-spread use of CASE tools.

***Provision of formal training in software engineering methods***: It was found that organisations, which provided formal training in software engineering methods, were more likely to use CASE tools than organisations which did not provide formal training in software engineering. In addition, organisations which provided formal software engineering training were more likely to have engaged external consultants to assist in the implementation of software engineering methods and tools. These findings support the observation made in Sumner's paper (Bergin 1993) that "CASE tools do nothing unless you understand and apply the underlying principles of software engineering".

## 2.6 Survey of important OO tools :

### OODesigner

OODesigner is a CASE tool for Object Modeling Technique (OMT) (Taegyun 1998). This product has two types of goals, product goals and process goals. The product goals are the functional requirements of OODesigner. The requirements included the following:

- Support for three models of OMT
- Documentation for class resources
- Checking consistency between objects within diagram
- Maintaining information repository for object model
- Code Generation for C++
- Reverse engineering for C++
- Storing/retrieving class definitions for reuse
- Collecting metrics data for C++ program

The qualitative process goals are:

- Improve the ability to conduct object-oriented (OO) design and implementation activity.
- Practice seamless and iterative characteristics of the OO development process.
- Apply OODesigner as a CASE tool for developing itself.
- Ensure maintainability for further enhancement and platform migration.

In 1996, the version 1.x of OODesigner satisfied most of the product goals. This version was implemented with 60 thousands of delivered source instructions (KDSI) of C++ code. But the version did not satisfy the process goals, especially with respect to maintenance issues. In other words, the old version worked correctly for the given requirements, but it had bad class structure for enhancing functionality. Thus OODesigner has been restructured since mid 1996.

Several lessons were deduced from this project:

- It is inevitable for beginners of OO paradigm to fail in the first OO project even if they are experts of structured technique. They might implement operational software, but their system become harder to maintain as time passed.
- An OO project could be successfully conducted just in the case of applying OO methodology, OO language and OO CASE tool synergistically. Using, OO language alone without methodology to build OO software should bring fake OO software.
- If you feel the need of restructuring an OO legacy system, do not hesitate to restructure it. To defer restructuring will cause a critical maintenance problem that can not be avoidable sometime in the future.
- Ill-designed OO software makes maintenance activity terrible, but well-designed OO software makes it enjoyable. This fact says the importance of OO modelling and design.

**ObjectiF from MicroTOOL**

(http://www.meridian-marketing.com/OBJECTIF/op_oop.htm)

ObjectiF from MicroTOOL of Berlin, Germany, is an integrated software development environment that supports all development steps from analysis to implementation in C++. ObectiF has been conceived for project work over LAN's. It has its own object base that guarantees reliable usage in a multi-user environment.

**OOA and OOD Practical Methods**

ObjectiF is based on the object-oriented analysis and design methods (OOA and OOD) of Coad/Yourdon because they are simple and practical. They offer an intelligible and easily understood graphic notation for classes, their properties, and the relations between them. ObjectiF has contributed three basic OOA/OOD method additions: an expressive tool for the description of message flow; object state transition diagrams for specifying the life cycle of objects; and the new concept of a subject that forms the basis for the reuse of class definitions beyond the confines of a given project.

**Mastering the Multitude of Classes with ObjectiF**

The object-oriented approach is automatically associated with reusability - with respect to ObjectiF, correctly so. But you can only reuse those classes that you can find again. That's why ObjectiF groups class models together in larger model units - we call them subjects. A subject will usually contain 20 to 30 classes that all deal with the same problem-domain or technical topic, and partake in an extensive exchange of messages. You can specify each class in a subject with the public characteristic, making it possible to reuse that class in another subject, or you can

specify it as private, prohibiting its use elsewhere. The public classes make up a subject's interface. That makes it possible to tell with a glance which classes of a subject are available for reuse.

Consequently, ObjectiF enables the practical reuse of classes even, or especially, beyond project borders.

ObjectiF is equipped with a ready-made subject designed as an efficient link to the Microsoft Foundation Class Library (MFC). lt contains the MFC classes together with their public methods. You will immediately be able to use them in the subjects of your application. Use means here, for example, developing your own user interface by creating a subclass of CDialog, or incorporating Cstring and CTime, as atomic types, in your attribute definitions.

**ObjectiF : the OOP Specialist for C++**

OOP in C++ means supplying a class declaration for every specified class, and coding function definitions for every specified method. What does ObjectiF do in all this? A whole lot. This is where OOA and OOD really pay off with regard to productivity and the quality of the end result.

Let's start with the class declarations: ObjectiF can represent every graphically specified class, with its attributes, methods, and method parameters, in the syntax of a class declaration with member list. You will only have to extend the member list with those C++ declarations that were not taken into account during OOA and OOD. If necessary, for example, a data member can be completed with a pointer operation, or the member list can be extended with declarations for enumeration and friends.

ObjectiF can view the relations in a class model from a C++ perspective too. It generates two things for a structural relation between classes. For one, it generates attributes for the assignment of those instances' object identifiers with which the class objects are associated over the relation. For another, ObjectiF generates methods that automatically safeguard the integrity of the relation expressed in the object identifiers.

**ObjectiF Stands for Accurate Specification Implementation**

The graphic specification and the class declarations are simply two different views of the same modelling objects. When you modify, add, or delete a method or attribute in a class model as an after-thought, you will find that the corresponding modifications are immediately made in the class declarations. The same thing is true at the relation level: If a structural relation is modified, added, or deleted in a class model, the data and function members are modified accordingly.

**OOA/OOD with ObjectiF-the Pleasure of interactive Modelling**

A smooth transition from analysis to design is a characteristic of object-oriented procedures. The two steps differ only in the objects being modeled. OOA refers to the modelling of problem-domain classes. OOD is concerned with the development of base technology classes, from which - put simply - the problem-domain classes can inherit technical behaviour. The modelling tool common to both is the class model; it graphically presents all of the central aspects of an object-oriented design: the classes, their static properties, the attributes, their behaviour, the methods, inheritance hierarchies, message connections, and structural relations at the instance level.

A class model contains those classes necessary for realizing a common problem-domain, or technical task. Because a class can contribute to the realization of several tasks, class models will often be redundant. Still, as an ObjectiF user, you can always rest assured that you are dealing with the current state of a class - this is, of course, even true in a multi-user environment.

What conditions can cause a class instance to display its behaviour? ObjectiF offers a special modelling tool for the specification of this aspect: the object state transition diagram it illustrates the connection between the following: the different instance states; the events that affect them, in the form of messages; the methods that are thus invoked, and the successive states that are then reached.

**A Maintenance Plus: Readable, Expressive C++ Code, ...**

...and that's guaranteed to be the end result of OOP with ObjectiF. During OOA/OOD, names are given to the class instances from the different perspectives, and each one represents a role from the problem domain. During the naming process, ObjectiF displays all the names used thus far for the class instances. Thus, you can be sure that instances in the same role are named the same - even when you later change a name - because ObjectiF ensures consistent use.

The meaningful given names are used by ObjectiF in the function definitions as names for variables - wherever necessary, they will be slightly modified to conform to C++ syntax. Through the use of these names in the definitions for formal parameters, local variables, and method parameters, each and every statement in the function definitions is easily understood without having to explore

the code to find the hidden meaning. That's why the code remains a readable representation of the problem-domain specification.

**Getting the Code to the Compiler**

The result of your work with ObjectiF will be compilable source code. When you define, for each subject, which class declarations are assigned to a header file, and which function definitions to an implementation file, then, ObjectiF will generate from that compilable .h files .cpp files, together with the include and forward declarations. When you test and debug outside ObjectiF, you can immediately make the necessary corrections. You tell ObjectiF to reunite the corrected code with its specification with the press of a button.

**The Wizdom Application Generator**

(http://www.meridian-marketing.com/WIZDOM/index.htm)

Very few user-friendly visual tools for the development of user applications complying with the Object-Oriented Model (OOM) presently exist. Although most of today's Fourth Generation Languages claim to have various Object-Oriented capabilities, as far as is known, at this time, none of these products has facilities required to define an end-user application as object-oriented in terms of the OOM, as does Wizdom Application Generator. Currently available 4th Generation Languages are visual, user-friendly extensions of conventional programming design and implementation concepts.

To date, true object-oriented user applications have been written in specialized lower-level Third Generation Languages such as C++. The use of these languages can be time-consuming, require specialized and costly programming personnel, and is often cumbersome.

The Wizdom Application Generator is a vehicle through which object-oriented user applications can be developed with ease and simplicity of a Fourth Generation Language. For a 4th Generation Language to build true object oriented end user applications, it must have the facilities required to define the application in terms of the object-oriented model. It must be a visual, user-friendly extension of object oriented programming design and implementation concepts.

**Wizdom Application Generator Facilities**

As the Class Manager defines Classes and Subclasses, it automatically generates the application's object-oriented database structure. The Form Generator

and Report Generator subsequently generate the database access plan. Database services are requested using the DBL (Dialog Box Language), where all requests are automatically planned and implemented. Linking data items between Classes, the MDI manager creates automatic access relationships between Class data.

**Class Manager**

The Class Manager (CM) provides the mechanism for definition of a complete end-user application in terms of object-oriented methodology. The entire structure of an application is defined, viewed, and subsequently modified, via the Class Manager.

On the left hand side of the CM screen appear the classes and subclasses representing an application; on the right hand side appear the members (procedural methods, forms, menus, reports, etc.) related to (encapsulated in) each class and subclass. One specifies object-oriented relationships between classes such as inheritance, and can zoom in on a class-related member to view or modify its definition.

**Form Generator**

Via the Form Generator, a form is associated with a specific class/subclass defined in the Class Manager. The data elements of the class/subclass are placed (dragged and dropped) directly onto the screen. One can simply point and click to specify element properties such as fonts and colours. Using the form, one can browse and edit the user data related to the class. The form is a window to the user data belonging to the class.

The Wizdom Form Generator enables one to build comprehensive GUI

windows with the entire range of functionality provided by Microsoft Windows and more. The specification of forms is accomplished by visual toolbox, point-and-click functionality.

Forms can be automatically initiated by predefined system events (i.e., pushing a button, database activity, etc.). Also, one can imitate a user-written procedure (or a system function) on system-predefined screen events.

In addition, the menu-generator option allows one to easily create menus for the form and/or the application desktop. The menus created operate in pop up/pull down mode, and menu items can specify automatic initiation of a user procedure, activate or produce a report, or perform a system function.

**Report Generator**

With the Wisdom Report Generator, reports are associated with a specific class/subclass defined in the Class Manager. Furthermore, reports can present data belonging to multiple associated classes.

Creating complex reports is facilitated by Wizdom's visual toolbox, point-and-click functionality. The Report Generator has object-oriented capabilities up to now rarely available. For example, the structure and format of a report can dynamically change with the class/subclass of data being displayed. For example, in a report of all employees displaying basic employee information such as employee number, and name, additional fields can dynamically appear on the report page, based on the type of employee (i.e., subclass) being processed.

So that, if the employee is a manager, his/her report data and format will automatically display manager-specific information, which will differ from the

report data and formats displayed on the printout for secretaries, programmers, etc. This is a classic example of a "polymorphic" report.

**Dialog Box Language**

For writing methods which easily incorporate simple or complex business logic into an application, Wizdom provides a Pascal-like Dialog Box Language (DBL), based on easy fill-in-the-blanks technology: one simply points and clicks the desired operations which appear in the dialog box window.

Using DBL, methods (application procedures) are developed which are associated with (encapsulated in) a specific class/subclass defined by the Class Manager.

Wizdom also provides a free-form format for Pascal-like procedures with the Wizdom Pascal Editor. Here free-form code is aided by visual tool-bar, point-and-click functionality.

**Multiple Document Interface (MDI) Manager**

The MDI Manager provides the capability to design application desktops containing multiple open forms; application end-users can work concurrently with different application-related forms on the same desktop. Forms presented on the application desktop can belong to the same class or different classes.

Also, as described below, one can utilize unique MDI Manager options to create automatic processing relationships between forms presented on application desktops.

By using the automatic scrolling option, one can specify that one or more forms on the desktop, belonging to the same class, automatically display the same

occurrence within the class. For example, a desktop can contain three forms, which display varying views of employee data. Whenever a user scrolls to a different employee using any one of these forms, the other two forms will automatically scroll to display information relating to the same employee.

One can create links between data items belonging to different classes. With data links, displayed forms, belonging to different classes will, automatically scroll to display data items linked together.

For example, if employee and project assignment data have been linked together, as one scrolls through the employee form, the project form will automatically display the project assignments which the employee is currently working on.

Using the polymorphic form option allows the structure and format of a form to be dynamically changed based on the class/subclass of data currently being displayed. For example, a form which displays all employees presents basic employee information. Additional fields can then dynamically appear on the form, based on the type of employee (i.e. subclass) being processed. Therefore, if the employee is a manager, his form data and format will automatically display manager-specific information, differing from data/formats displayed for secretaries and programmers.

## 2.7 A Software Maintenance process Model:

## GUMP (Generic University of West Florida Maintenance Process)

It is now generally accepted that the first requisite for improving the timeliness and quality of an organization's software products is a mature software process. The establishment of a defined software process is, in some markets, becoming a requirement of doing business.

The new emphasis on process creates challenges for both software engineering educators and for software development organizations, and unfortunately there is still little published information to guide them. 'Process' is a very difficult subject to discuss in the abstract. Educators teaching about process need to have a range of processes to which they can point. Managers establishing a process in their organizations likewise need example processes as starting or reference points.

But examples of successful, fully elaborated processes are hard to come by. Most companies that have expended the effort to define and validate their own software process regard the results as proprietary, and well elaborated processes for use in the classroom are still very scarce .

The overall structure of the GUMP process architecture (Wilde and Brown, 1996) is shown in Figure 1. The process is driven by *Change Requests* submitted by either customers or team members suggesting improvements or bug fixes in the software system. *Change Requests* are given a brief sanity check and, if they pass, filed as *Deficiencies (cell* 100). When resources are available, the Project Co-ordinator selects one or more related *Deficiencies* for analysis, thus initiating a *tisk,* as a complete cycle of software change is called (cell 200).'

A *tisk* starts with an analysis step (cell 300) which defines requirements and high-level design for the software change, along with a risk analysis and an estimate of the resources needed for implementation and testing. The final product of this cell is an *Analysis Report* which is subjected to an inspection supervised by the Software Quality Assurance team.

The approved *Analysis Report* then goes to the Change Control Board (CCB), composed of the Client, the Project Co-ordinator, and one member each from Independent Verification and Validation, Software Quality Assurance, Software Configuration Management and Software Engineering (cell 400). Here the basic decision to commit resources to the change is made. If the decision is positive, Software Configuration Management allows the Software Engineers to cheek out any code and needed documents from the configuration management system (cell 500).

Change implementation (cell 600) consists of making the changes to code and documents, unit testing, and a final inspection again supervised by Software Quality Assurance. The work then goes to Independent Verification and Validation, which consists mainly of integration and system level testing (cell 700). If testing is successful, the revised software is checked back in to the configuration management system (cell 800).

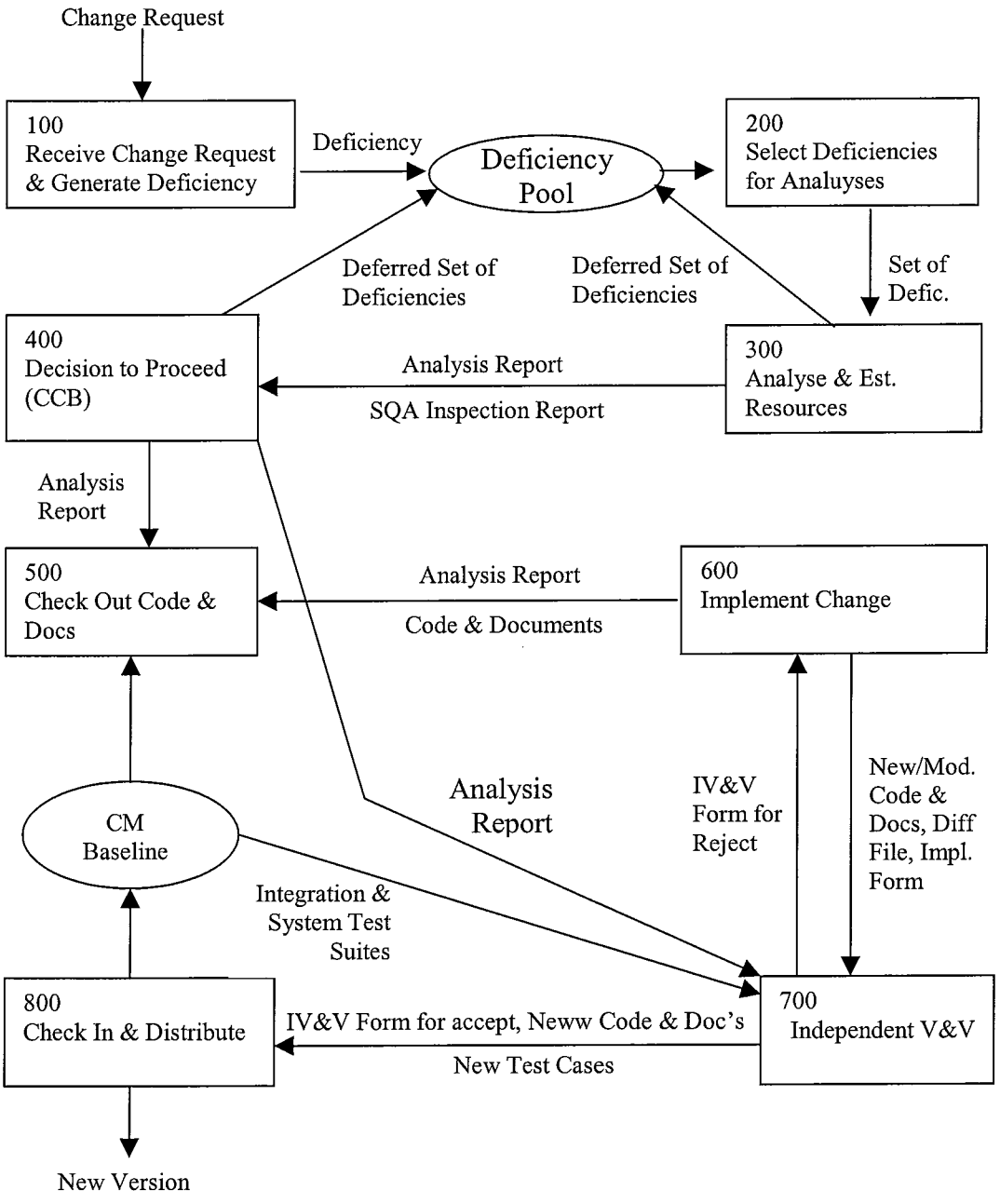



Figure 1. Overall structure of the maintenance process

Most of the high-level process cells shown in Figure 1 have been further broken down into more detailed cells.

At the lowest level, each cell is described using an 'Entry-Task-Exit (ETX)' format adapted from that suggested by Watts Humphrey (Humphrey, 1989). As an example, Table 6 shows the description of cell 601, *'Unit Testing'*. The tasks for the cell are briefly listed and the group responsible (Software Engineers, SQA, etc.) is identified. Cell task descriptions may reference GUMP standards documents that provide detailed guidance on how a task is to he carried out. The unit testing standard, for example, states that the 'Automatic Test Analysis for C (ATAC)' coverage tool (Horgan, London and Lyu, 1994) should be used to check that every testable block and decision is covered, and that non-testable decisions, such as tests for operating system errors, should be explicitly hand checked. GUMP also includes standard forms to be used in most of the main tasks.

Additionally, certain cells have associated metrics that are collected, along with weekly timecard data, and kept in a central database, available to all team members. The data are intended to be used to make better and more accurate time estimations and schedules *for tisk* milestones. Finally, some cells have required training that must be completed in order to accomplish the subcell tasks.

Table 6 - Entry-task-exit description of the 'unit test' cell

| Cell | 602 - Unit test |
|---|---|
| Entry (from cell) | -(601) New/modified code and does |
| Exit (to cell and group) | -(603) New/modified code and does<br>-(603) New tests (as required)<br>-(603) Diff file<br>-(603) Implementation form |
| Feedback in | -(603) Testing rework |
| Feedback out | -(601) Implementation rework |
| Tasks (responsible group) | - Conduct unit testing IAW testing std. (SE)<br>- Generate diff file and implementation fm. (SE)<br>- Attach diff file and detailed design to implementation fm. (SE)<br>- Inform PC ready for inspection (SE) Measures |
| Training required | - Project orientation, ATAC |

## 2.8 Software Maintenance CASE Tools

The software maintenance support tools provide support for program understanding. These tools assist the programmer/analyst in discovering the physical and logical designs of the system at hand. The discovery of impacts of proposed changes on "distant" programs, i.e., programs linked by common data elements, is an important part of this phase, and is assisted by these software tools. These tools may also contain functionality that facilitates the coordination of programmers on a large project. Such functionality is provided by a variety of schedule and project management programs and communications programs such as email, audio and video conferencing. We study software maintenance tools and their usage because maintenance is a critical MIS task and thus, tools that adequately support this task potentially could provide significant value to organizations. To add value, information technology (IT), e.g., software maintenance tools, must meet the needs of the organizations, groups, and individuals who use it. This truism is as old as IT and the MIS field. It is embedded in the systems analysis process, especially in the requirements determination and analysis step (Dishaw and Strong, 1998).

### Task-Technology Fit

A fundamental argument is that software will be used if the functions available to the user support the activities of the user. A software function supports an activity if it facilitates that activity. Alternatively, the software must serve to lower the cost to the user of performing the activity. Rational, experienced, users will choose those

tools and methods, which enable them to complete their activities with the greatest net benefit. Software which does not over sufficient advantage will not be used.

## Maintenance task activities

Vessey, during protocol analysis sessions (Vessey, 1986), developed a description of the actual types of actions engaged in by all maintainers. She identified Planning, Knowledge Building, Diagnosis, and Modification Activities in the maintenance process. These are the set of actions performed by maintainers to change existing software. The first three cover understanding, while the last one is the actual program transformation activity.

In addition to understanding and modification activities, which are the core activities of the maintenance task, coordination activities are necessary (Vessey and Sravanapudi, 1995). In most MIS organizations the programmer initiates a production release process which may include documentation updates and testing for standards adherence.

Although coordination activities are normally a small part of the maintenance task, such activities ultimately have a significant impact on the success of the maintenance project. These coordination activities were not found in Vessey's protocol analysis because her experimental task ended at production release.

## Maintenance tool functionality

Henderson and Cooprider (1990) provide a description of the basic functions present in design support software (CASE). They identified two major dimensions of

tool functionality: Production and Coordination functionality. Production functionality is functionality that supports an individual programmer developing or changing software. It includes representation, analysis, and transformation technology. Representation functionality helps the programmer in representing the problem and thus aids in understanding the problem. Analysis functionality supports exploration and evaluation of representations, and thus aids in building further knowledge and understanding about the problem, diagnosing problems, and planning solutions. Transformation functionality supports the actual changes or additions to software.

Coordination functionality is functionality that supports the coordination activities necessary when an individual performer is working in an organization. It includes control functionality to ensure that programmers are following standard procedures and cooperation functionality to support interactions with others.

The support of the software maintenance process, especially the program understanding and modification portion, can be viewed as a problem of supporting representation development, manipulation, and testing. At the heart of the process of understanding a program is building a representation of the problem, that is, a mental model or conception of the problem to be solved. Problem representation is an important part of the problem-solving process. Thus, the development of a program representation is an essential part of the software understanding process.

Program understanding may be viewed as the process of recognizing plans or intentions of the code and is essential to the completion of a maintenance task. The understanding process depends upon the programmer developing representations of

program elements, manipulating and integrating these elements, and testing the result for correctness. This process is iterated as necessary. This process is more difficult when the plans are delocalized or spread over the module, or even between modules, or when a program interacts with other programs in non-obvious ways. Software maintenance support tools may assist the programmer, in part, by mitigating the problems associated with delocalized plans.

Some success has been achieved in the creation of tools that support program understanding. These tools support the development, manipulation, and testing of representations of the application software, and are able to produce higher level abstractions from code in the form of a variety of structure and flow charts. In addition, dependency analysis is possible using these tools. These tools support the programmer in the development of an understanding of a program through the generation of a representation of a program's functions and data structures. With these tools, the programmer can test his or her mental representation through analysis of a program by "single stepping" and displaying the contents of variables.

Program understanding, however, is not the entire story of software maintenance production support. The programmer must be able to actually change software and document the effects of that change. Program modification is intermixed with the understanding process. Typically, the programmer arrives at a point where he or she is about to test an assumption (representation) about the software to be changed. The task being performed in this process is diagnosis. In diagnosis activities,

a hypothesis is tested and confirmed or rejected. A change can be made to test the behavior of a program.

The actual change process is fairly simple. It includes making a change in a source module, compiling the module, and testing the changed program. The software that supports it consists of an ordinary text editor and library management software. The compiler program also falls into this category.

Coordination fit includes support for cooperation and coordination among programmers as they plan and release modified software as well as controls to ensure conformance to software standards. Case tools are important in the support of the coordination efforts of systems professionals. Current CASE tools are generally designed for individual use but provide some support for coordination among programmers (Vessey and Sravanapudi, 1995). Programmers also typically have access to a number of other software technologies that support their work. These tools include project management software, groupware, and a wide variety of software engineering tools that are not specifically labeled "CASE" tools.

In the area of control (compliance with standards and practices), software tools vary in the support they provide for enforcing methodology standards and processes. The development and adoption of more modern maintenance tools is addressing this problem. Project management software facilitates management control over software development and maintenance projects. The newest project management packages integrate with CASE tools and facilitate both tracking and reporting activities. Software which supports coordination activities includes testing and version (release) management software. Software which supports programming standards aids the

coordination task as well. Project management software that also facilitates coordination of maintenance activities that involve dependencies with other activities is also included in this category.

An increasingly common trend in the management of maintenance is the distribution of maintenance activities to outsourcing organizations, some of which may be "offshore" (Kumar et al., 1996). Wide area networks such as the Internet can be used to communicate with these vendors. The software used in these situations may include e-mail, file transfer (FTP), and conferencing, including chat. Recently, the internet has come to support video conferencing. Software for this application typically supports, in real time, exchange of video, audio, and graphics (whiteboard). Even when offshore outsourcing is not employed, the increased use of distributed teams as well as the increased incidence of telecommuting have driven the adoption and utilization of these technologies. Software facilitation of coordination activities has thus become much more common, although coordination tool functionality has not yet reached the levels of sophistication or integration found in production functionality.

## 2.9 Example of a Change Request and Problem Reporting Process

A change request represents a documented request for a change and an associated process model for change. LIFESPAN, an MCS tool, models the change request via a series of "forms" and the process of change via a series of states, tasks and roles (Brown, A., Dart, S., 1991). A customer may submit an on-line Software Performance Report (SPR) which identifies a fault or a request for an enhancement for versions of components. This allows the report to be investigated by circulating it to the original designers and implementers who can diagnose the problem. In response to the SPR and change impact analysis, an on-line Design Change (DC) is proposed. This details exactly what components are to be changed and how. LIFESPAN analyses who would be affected by the change. Those people are then automatically chosen to be the Change Control Board. They are notified by electronic mail about the DC and must vote within a certain time frame on whether to approve the change. Once the DC is agreed to, a new development version of the code to be changed is made, the DC's state becomes "active" and the code to be changed is locked. Upon completion of the changes, the new version is frozen and submitted for checking and approval to a person with QA privilege. Upon approval, the code changes acquire an "approved" status, the status of the DC becomes "approved" and affected users are notified by electronic mail that the new version is available. The users are notified via a Software Status Report (SSR) which closes off the original SPR. Thus, the SPR, DC and SSR not only provide a means for users and maintainers to communicate, but they also represent a history of changes related to a particular change request; status reports for changes in progress; audit trails of changes completed; a supporting mechanism for change impact analysis and ensuring that the

appropriate people carry out their tasks at the right time. In effect, change requests assist in driving the process of change.

# Chapter 3 - Tool's Specifications

The objectives of this tool are to provide guidelines and procedures for carrying out a variety of maintenance activities on Object Oriented Softwares.
A change control framework, around which software configuration management disciplines are applied, aims to systematise the software maintenance process. This is done, by specifying the chain of events and the order of stages that a proposed change has to go through. The outcome of each stage is represented by forms, which allow a methodical approach to the establishment and control of traceability throughout the maintenance process.
These forms could be also the source of documentation of maintenance history and system redocumentation to improve the future maintainability of the software systems being maintained and the ease with which changes can be accommodated.

The outcome of each phase in the maintenance framework, is a form, which offer objective visualisation of the evolution of the maintenance process. Each form, consist essentially of three sections: identification, status and information.

The software maintenance framework stages are:

1. Change request
2. Change evaluation
3. Maintenance specification
4. Maintenance design
5. System release

Moreover a set of attributes for communicating the meaning of the problems and defects found are also necessary for maintaining OO software systems. Some of these attributes are :

1. identification and description of a problem
2. uniqueness of a problem
3. date and time of a problem occurrence
4. criticality and urgency of a problem
5. classification of problem causes(defects)

For representing problems, problem reports can be issued where they are uniquely identified and described. The points of information relevant for describing problems are :

- description of observations made during a problem occurrence
- effect and consequence of a problem
- temporary actions taken to circumvent
- suggestions for improvement
- connections to other problems

As for the Criticality and Urgency attributes, the former defines the seriousness of the disruption caused by a problem. The later communicates the significance of an immediate corrective measure. High criticality often implies high urgency.

# Chapter 4 - Tool's Design

The Object Oriented Maintenance Management (OOMM) model aims at improving activities by providing guidance throughout the maintenance process, and determining the organisation and content of the information needed to support these activities.

The model is based on the traditional waterfall life-cycle model of software development. This is a convenient approach, because it allows the process to be represented in a graphical and logical form providing a framework around which quality assurance activities can be built in a purposeful and disciplined manner.

Each OOMM phase is defined in terms of the output produced during the phase. The outcome of each OOMM phase is a form which expresses a point in the maintenance process. These completed forms are, therefore, the natural milestones, i.e. the baselines of the software maintenance process, and offer objective visualisation of the evolution of that OOMM model.

Figure 2 represents the OOMM model. The rectangles in the figure represent OOMM phases and the ovals represent the baselines formed from the output of the phases.

Because it is a software maintenance model, it is essential that the influence of the existing software system on the process should be represented. It is for this reason that the *change evaluation* phase has been introduced in which modifications are considered in relation to the existing software system.

As with models of software development, OOMM phases may overlap. Also, it may be necessary to repeat one or more steps before a change is completed.

However, the products which represent the output of the phases must constitute a baseline and cycling must be controlled.

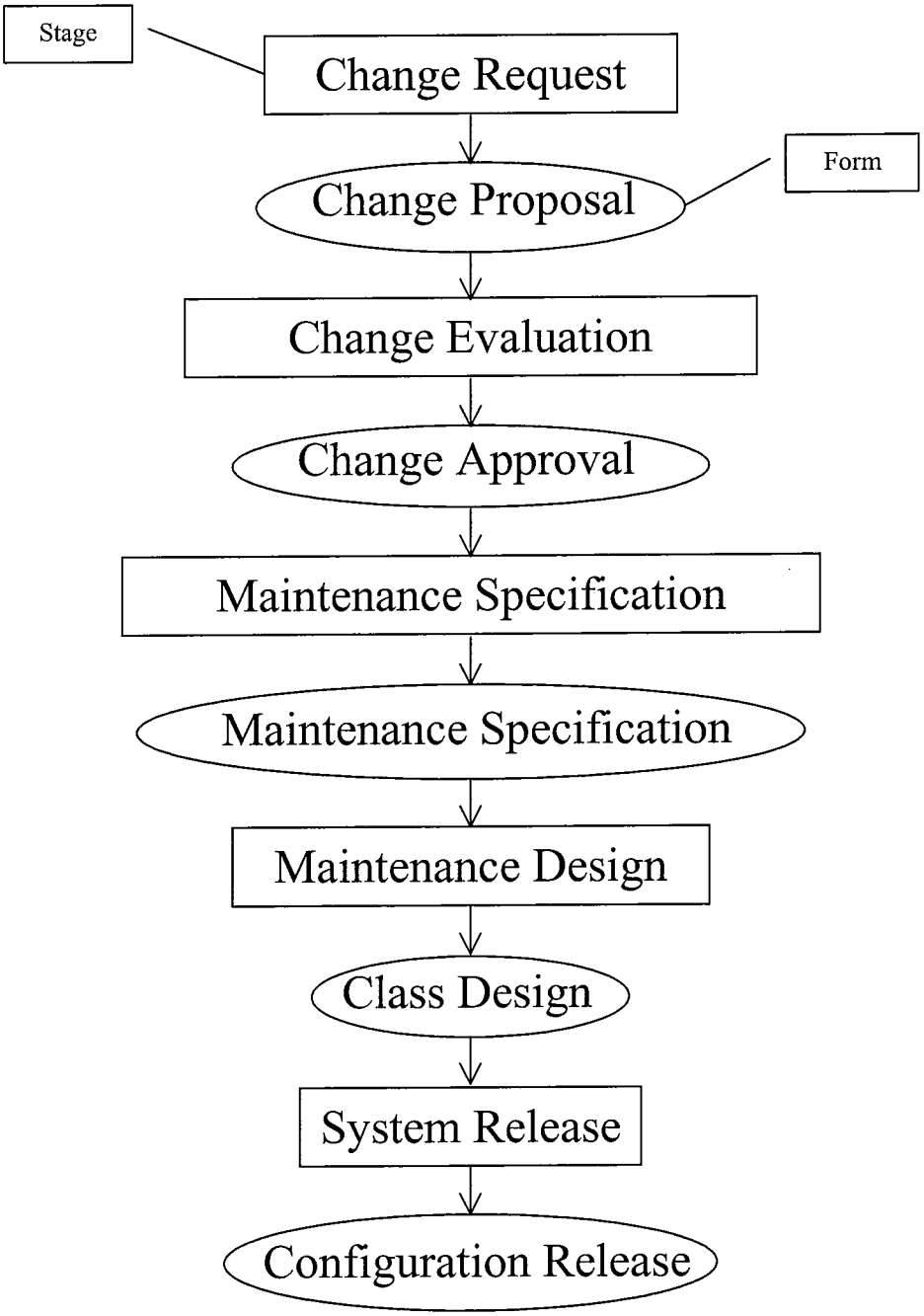



Figure 2

The following subsections discuss each of the OOMM phases in greater detail.

- Change Request

All requests for software maintenance will be presented in a standardised manner. The Change Proposal form is the form associated with this OOMM phase. The completion of a change proposal form triggers the process of maintenance. The form contains the basic information necessary for the evaluation of the proposed change. If the proposed change is for the corrective maintenance, then a complete description of the circumstances leading to that error must be included. For other types of maintenance, an abbreviated requirements specification must be submitted.

- Change Evaluation

In the *change evaluation* phase, the maintainer is primarily concerned with understanding the change, and its effect within the software system. An accurate change diagnosis is performed to assess the feasibility of the proposed change in terms of cost and schedule resulting in approval or rejection. A rejected proposed change is then abandoned. If the proposed change is approved then a corresponding Change Approval form is created. The Change Approval form is one of the documents used as the basis for planing the system release. It is a vehicle for recording information about a system defect, a requested enhancement or quality improvements. The change approval form along with its corresponding Change Proposal form, is the basic tool of a change management systems. By documenting new software requirements or requirements that are not being met these forms become

the contract between the person requesting the change and the maintainers who work on the change.

In this phase, the work required by the proposed change is classified as perfective, adaptive, corrective or preventive maintenance. In addition, every software component involved in the proposed change must be known. The inadequacies, or unfulfilled requirements described in the Change Proposal form are identified in the existing software system. This identification involves different aspects of software which depend on the type of the change required.

- Maintenance Specification

This phase is characterised by the structure of the modification, which is in the form of a complete, consistent and comprehensible common specification of all the changes proposed and approved for a planned scheduled release. In addition, how the software classes have to be modified needs to be clarified. The resultant form for this phase is the Maintenance specification form which is generated after having selected the approved changes for the next system release. The design of a modification requires an examination of the side-effects of changes. The maintainer must consider the software classes affected and ensure that component properties are kept consistent. Additionally, if the changes require a new logic or new features to be added to the system, then these have to be specified and incorporated. In the specification of the proposed change, different aspects of the system should be considered which depend on the type of maintenance required.

- Maintenance Design

This phase facilitates system comprehension by incremental redocumentation of the existing software system, as proposed by the method. The form will be filled in when the corresponding software classes have to be modified. During this phase the algorithms and the behaviour of procedures for both normal and exceptional cases are explained. In addition, the tests for each of the changed or implemented software classes and/or methods are planned. The form associated with this phase is the class design form.

- System Release

System Release is the last phase of the OOMM before a new configuration containing the approved changes is released to the user. Validation of the overall system is achieved by performing the integration and system tests on the system. Once modification on the system have been performed under the configuration control function, the task at this stage is to certify that all baselines have been established. The Configuration Release form contains details of the new configuration. A configuration Release form is the software system release planning document , which aims to keep the information pertaining to the history of a maintenance phase.

All of the above mentioned OOMM phases are closely controlled by a user profile scheme The user profile concept is designed to control access permissions not only to OOMM activities and phases but also to any task related to products, parameters tables, queries and reports within the entire application. Update and delete permissions are allocated to user profiles for the different available functions. User

profiles are given user ids' and passwords to restrict and control access to the application in the first place and from then to various existing screens.

Moreover, products notion is essential to give OOMM model its practical implementation ease. A predefined product (application, software) is developed in at least one platform and eventually has at least one release. Many platforms could be associated to one product and each product/platform combination might have different releases.

# Chapter 5 - Tool's Implementation

The OOMM tool is implemented in Visual Basic 5.0 under Windows 95. VB offers a friendly and intuitive interface through its extensive event-driven language, its object-based structure and its support for OLE. VB provides a fast dabase engine (Jet) and a wealth of new features and methods of controlling and accessing the data. The Jet engine can create and manage information in a wide variety of database formats. Jet can deal with Access, Foxpro, DBase, Paradox, Oracle, SQL Server and Btrieve databases. OOMM tool uses Access 97 as back-end database (oommt.mdb) where all transactions and operations are stored along with predefined queries. The VB front-end presents these information for the user in different set of views and in a very simple way.

All information related to all OOMM phases are stored in tables within the end-back database. These tables along with other codes tables, user tables and product tables listed in the following table (Table 7) are discussed later in greater detail.

| Table Name | Description |
|---|---|
| TL_Trans_H | Transactions Table Header |
| TL_Trans_D | Transactions Table Details |
| TL_Trans_Class | Transactions Table of involved classes and methods |
| TL_Product_H | Products Table Header |
| TL_Product_D | Products Table Details |
| TL_User | Users Table |
| TL_User_Func | Permissions/Functions Table allocated for users |
| TL_Test | Tests Table conducted on involved classes + methods |
| TL_Function | Functions Table |
| TL_Tab_Class | Class Codes Table |
| TL_Tab_Method | Method Codes Table |
| TL_Tab_ModType | Modification Type Codes Table |
| TL_Tab_Platform | Platform Codes Table |
| TL_Tab_Priority | Priority Codes Table |
| TL_Tab_Status | Status Codes Table |
| TL_Tab_TestStatus | Test Status Codes Table |
| TL_Tab_TestType | Test Type Codes Table |

The follwing tables (Table 8, 9 amd 10) present respectively all the fields name, type, and description of TL_Trans_H, TL_Trans_D, and TL_Trans_Class data tables.

Table 8 - TL_Trans_H

| Field Name | DataType | Description |
|---|---|---|
| TH_Id | AutoNumber | Transacrtion Id (Unique Number Automatically Generated) |
| TH_Trans_Type | Text | Transaction Type (It's a code indicationg whether it's a Proposal, an Evaluation or a ...) |
| TH_Trans_Date | Date/Time | Transaction Date (The date at which the transaction is entered) |
| TH_Proposal_Id | Number | Transaction Related Proposal Id |
| TH_User_Id | Number | Transaction User Id (who is responsible of this transaction entry) |
| TH_Status | Number | Transaction Status (the current status of this specific transaction) |
| TH_Prod_Id | Number | Transaction Product Id (The Software/Product code being modified or updated) |
| TH_Platform | Number | Transaction Product Platform Code (On which platform the concerned product is developed) |
| TH_Release | Text | Transaction Product Release (at which Release the concerned product is ) |
| TH_Chg_Type | Number | Transaction Change Type Code (whether it's a corrective or adaptice or perfective or ...) |
| TH_Desc | Text | Transaction Description (What the modification or the transaction is about) |
| TH_Reason | Memo | Transaction Detailed Description (the detailed reason of the update) |
| TH_Priority | Number | Transaction Priority of Implementation (the urgency level of the modification) |
| TH_Conseq | Memo | Transaction Consequence (in case the update/modification hasn't been done) |
| TH_Chg_Spec | Memo | Transaction Change Specification (a +/- Detailed description of the technical modification) |
| TH_Int_Test | Yes/No | Transaction Integreation Test Flag (indicates whether an int. test is conducted or not) |
| TH_Sys_Test | Yes/No | Transaction System Test Flag (indicates whether a sys. test is conducted or not) |

The TL_Trans_D table keeps the status history of transactions.

| Table 9 - TL_Trans_D | | |
|---|---|---|
| Field Name | DataType | Description |
| TD_Seq | AutoNumber | Sequence Number To make it unique |
| TD_Id | Number | Transacrtion Id |
| TD_Status_Date | Date/Time | Date at which the transaction status changed from ... to ... |
| TD_From_Status | Number | The Status Code from which the trans. changed |
| TD_to_Status | Number | The Status Code to which the trans. changed |
| TD_User_Id | Text | The user id who changed the status |

The TL_Trans_Class table stores data about involved classes and methods in the modification.

| Table 10 - TL_Trans_Class | | |
|---|---|---|
| Field Name | DataType | Description |
| TC_Seq | AutoNumber | Sequence Number To make it unique |
| TC_Id | Number | Transacrtion Id |
| TC_Class | Number | The Class code involved in the update/modification process |
| TC_Method | Number | The Method code involved in the update/modification process |
| TC_Description | Text | The Description of the modification that should be done on this class or method |
| TC_Status | Number | The Modification Status Code (whether the mod is in progress, finished or not yet started) |

TL_Product_D and TL_Product_H data tables (Table 11 & 12) are related to products.

| Table 11 - TL_Product_H | | |
|---|---|---|
| Field Name | DataType | Description |
| PH_ID | AutoNumber | The Product Identification Code |
| PH_Name | Text | The Name of the product |
| PH_Desc | Memo | The Product Description (what it is about , what it does , ...) |

| Table 12 - TL_Product_D | | |
|---|---|---|
| Field Name | DataType | Description |
| PD_ID | Number | The Product Identification Code |
| PD_Release | Text | The Latest Version of this Product (VxRxMx) |
| PD_Date | Date/Time | The Date at which the latest release is done |
| PD_Platform_Code | Number | The Product Platform Code |
| PD_Class | Number | The Class Code (each product has many classes ; each class has many Methods) |
| PD_Method | Number | The Method Code |

The TL_User data table (Table 13) groups all necessary information related to users

| Table 13 - TL_User | | |
|---|---|---|
| Field Name | DataType | Description |
| TU_Id | AutoNumber | The User Identification Code |
| TU_Name | Text | The User full Name |
| TU_Login | Text | The User Login Name |
| TU_Password | Text | The User Password |
| TU_User | Text | The User who Created this New User or modiy it |
| TU_Dtm | Date/Time | The Date and Time this User is created or modified |

The following data table (Table 14) is a code table for the modification types. It is called TL_Tab_ModType. All code tables have the exactly same design.

| Table 14 - TL_Tab_ModType | | |
|---|---|---|
| Field Name | DataType | Description |
| TC_Entry_Code | Number | The Entry Code within a Table |
| TC_Abrev | Text | Abreviation or Alphabatecal Code (Up to 5 charac) |
| TC_Desc | Text | The Description |
| TC_Flag | Text | System or User Defined |

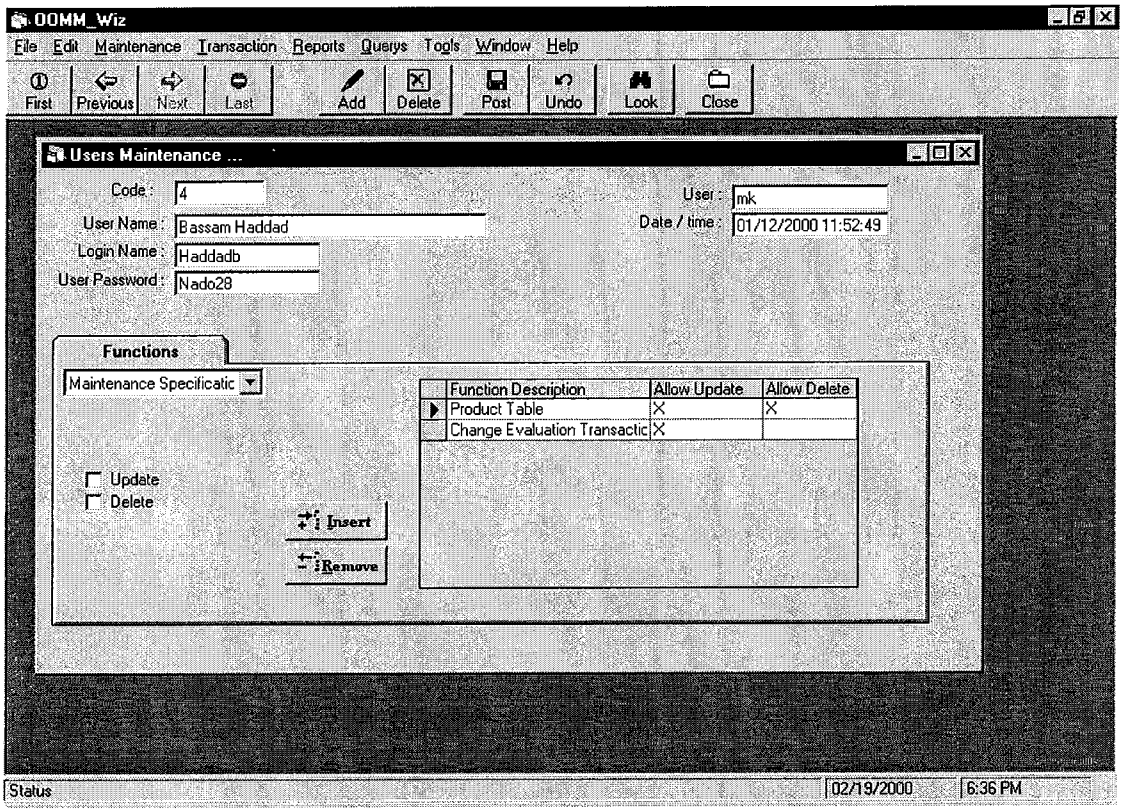
OOMM tool offers a set of queries and reports that brings essential and interesting features to users. Predefied queries select not only general information but also particular ones on various issues. For advanced users that have knowledge in SQL statement they can even formulate their own queries.

Reports that can be previewed on screen, printed, sent to Word file or to Excel sheet add valuable feature. For example Users may easaly investigate about any kind of transaction that has any status between two dates.

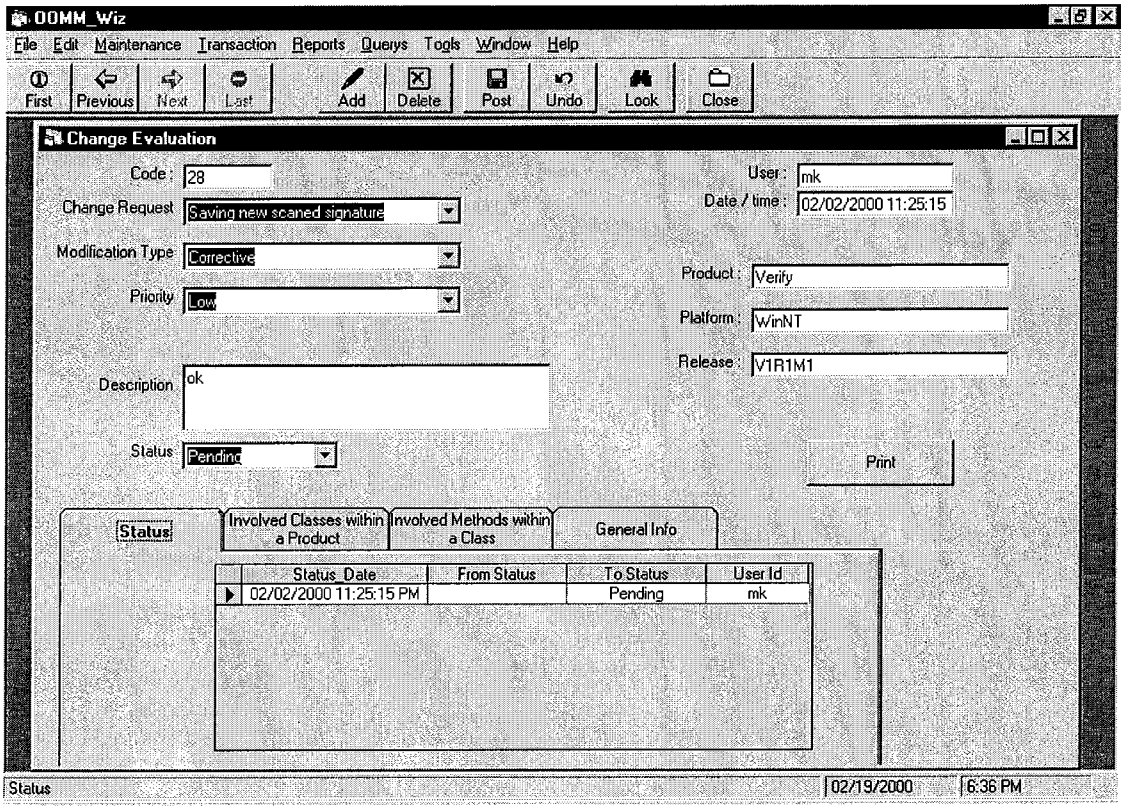
# Chapter 6 – Example Application

In what follows some snapshots of the application:

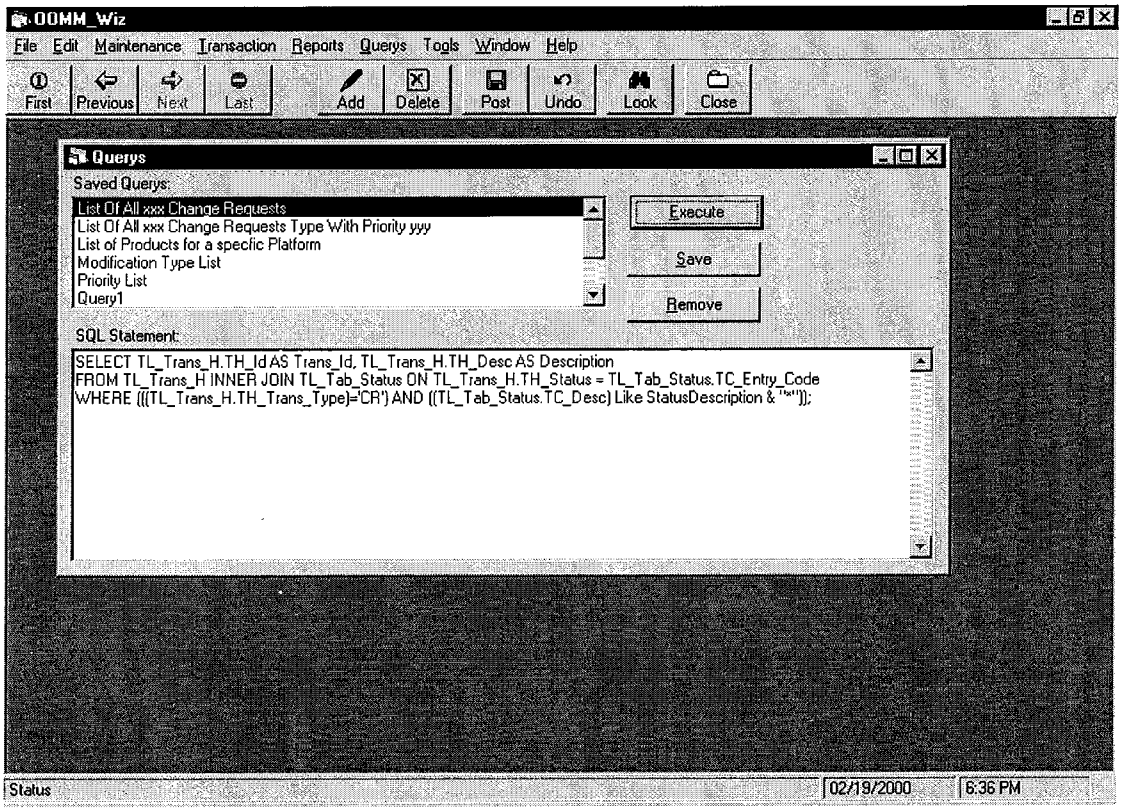
The User Maintenace Screen:



This is the change evaluation screen:



This is the queries screen:



# Chapter 7 – Conclusion

In this report a model for software maintenance management for object oriented systems has been presented. The maintenance management process was formally structured into a sequence of phases that conceived the backbone of the work. By defining this model, the steps, which a change should follow during maintenance, are clearly defined. The forms, which are the outcome of each phase, represent the maintenance history and an abstraction of the operational product necessary to improve the documentation of a poorly documented existing software system. This approach presents an increased flexibility and greater facility to apply the OOMM tool due to:

- The following of OOMM phases institutes a change control procedure to monitor changes.
- The completeness checks are assured through the use of forms since no essential details are omitted.
- The consistency checks are ensured through the use of forms since the information required by forms is provided by other forms in the configuration.
- The traceability between phases is facilitated by establishing in the forms the relationships between software components of different phases.
- The uniformity of information is guaranteed since forms are pre-defined thus avoiding inconsistency and unnecessary differences.

# Reference List


Bergin, T.J. (1993), *Computer Aided Software Engineering Issues and Trends for the 1990s and Beyond,* Idea Group Publishing.

Boehm, H. (1981), *Software Engineering Economics,* New York: McGraw-Hill.

Brooks, F. (1985), *The Mythical Man-Month,* Reading, Mass.: Addison-Wesley.

Brown, A., Dart, S., Feiler, P., Wallnau, K. ( 1991), The State of Automated Configuration Management, *Annual Technical Review, SEI.*

Connell, J., Shafer, L. (1995), *Object-Oriented Rapid Prototyping*, Englewood Cliffs: Yourdon Press.

Dishaw, M., Strong, D. (1998), Supporting software maintenance with software engineering tools: A Computed task-technology fit analysis, *The Journal of Systems and Software*, 44, 107-20.

Fisher, A.S. (1991), *CASE Using Software Development Tools,* New York: John Wiley & Sons.

Henderson, J.C. Cooprider, J.G. (1990), Dimensions of I/S planning and design aids: A functional model of CASE technology, *Information Systems Research*, 1, 227-54.

Horgan, J., London, S. and Lyu, M. (1994), Achieving software quality with testing coverage measures, *IEEE Computer,* 27, 60-9.

Humphrey, W. (1998), *Managing the Software Process,* Reading, Mass.: Addison-Wesley.

Joiner, J., Tsai, W., Chen, X., Subramanian, S., Sun, J., & Gandameneni, H. (1994), Data-centered program understanding, *Proceedings of the International Conference on Software Maintenance,* 192-98.

Kumar, M.P., Das, V.S.R., Netaji, N. (1996), Offshore software maintenance methodology, *Journal of Software Maintenance,* 8, 179-97.

Poo, D., Chung, M.K. (1998), CASE and software maintenance practices in Singapore, *The Journal of Systems and Software*, 44, 97-105.

Taegyun, K., Gyusang, S. (1998), Restructuring OODesigner: A CASE Tool for OMT, *International Conference on Software Engineering,* 449-51.

Vessey, I. (1986), Expertise in debugging computer programs: An analysis of the content of verbal protocols, *IEEE Transaction Systems*, 621-37.

Vessey, I., Sravanapudi, A.P. (1995), CASE tools as collaborative support technologies, *Commun. ACM,* 38, 83-95.

Wilde, N., Brown, S. (1996), The GUMP Process for Software Maintenance and Maintenance Education, *Journal of Software Maintenance: Research and Practice*, 8, 229-39.

http://www.meridian-marketing.com/OBJECTIF/op_oop.htm

http://www.meridian-marketing.com/WIZDOM/index.htm

# Appendix : Implementation Details

The following example is a code for the Login Screen:

```
Option Explicit
Public LoginSucceeded As Boolean

Private Sub cmdCancel_Click()
   'set the global var to false
   'to denote a failed login
   LoginSucceeded = False
   Me.Hide
   End
End Sub

Private Sub cmdOK_Click()
datUser.Recordset.FindFirst "tU_login = '" & txtUserName.Text & "'"
If Not datUser.Recordset.NoMatch Then
   'check for correct password
   If txtPassword = datUser.Recordset.Fields("TU_password") Then
     CurCode = datUser.Recordset.Fields("TU_id")
      sys = datUser.Recordset.Fields("TU_id")
      curuser = Me.txtUserName
      LoginSucceeded = True
      Set mywork = Workspaces(0)
      Set mydb = mywork.OpenDatabase("oommt.mdb", False, False)
      Set myuser = mydb.OpenRecordset("tl_user_func")
      OomMen011.Show
      Me.Hide
   Else
      MsgBox "Invalid Password, try again!", , "Login"
      txtPassword.SetFocus
      SendKeys "{Home}+{End}"
   End If
Else
      MsgBox "Invalid User Name, try again!", , "Login"
      txtUserName.SetFocus
      SendKeys "{Home}+{End}"
End If
End Sub

Private Sub txtPassword_KeyPress(KeyAscii As Integer)
        If KeyAscii = 13 Then cmdOK.SetFocus
End Sub

Private Sub txtUserName_KeyPress(KeyAscii As Integer)
        If KeyAscii = 13 Then txtPassword.SetFocus
End Sub
```

The following example is a code for the Maintenance Design screen

```
Dim TmpProdId As Integer
Dim TmpPlatform As Integer

Private Sub Refresh_MethodTest()
   vsql = ""
   vsql = "SELECT TL_Test.TS_Method, TL_Tab_Method.TC_Desc, TL_Tab_TestType.TC_Desc,
    TL_Tab_TestStatus.TC_Desc FROM TL_Test, TL_Tab_Method, TL_Tab_TestType,
    TL_Tab_TestStatus WHERE TL_Test.TS_Method = TL_Tab_Method.TC_Entry_Code and
    TL_Test.TS_Type = TL_Tab_TestType.TC_Entry_Code and TL_Test.TS_Status =
    TL_Tab_TestStatus.TC_Entry_Code and TS_Trans_ID = " & Val(CbRequest.BoundText)
   Me.DatMethodTest.RecordSource = vsql
   DatMethodTest.Refresh
End Sub

Private Sub Refresh_ClassTest()
  vsql = ""
  vsql = "SELECT TL_Test.TS_Class, TL_Tab_Class.TC_Desc, TL_Tab_TestType.TC_Desc,
    TL_Tab_TestStatus.TC_Desc FROM TL_Test, TL_Tab_Class, TL_Tab_TestType,
    TL_Tab_TestStatus WHERE TL_Test.TS_Class = TL_Tab_Class.TC_Entry_Code and
    TL_Test.TS_Method = 0 and TL_Test.TS_Type = TL_Tab_TestType.TC_Entry_Code and
    TL_Test.TS_Status = TL_Tab_TestStatus.TC_Entry_Code and TS_Trans_ID = " &
    Val(CbRequest.BoundText)
  Me.DatClassTest.RecordSource = vsql
  DatClassTest.Refresh
End Sub

Private Sub Refresh_InvMethod()
  vsql = ""
  vsql = "SELECT TC_Method, TC_Desc From TL_Trans_Class, TL_Tab_Method Where TC_ID = "
    & Val(CbRequest.BoundText) & " and TC_Method = TC_Entry_Code"
  Me.DatInvMethod.RecordSource = vsql
  DatInvMethod.Refresh
End Sub

Private Sub Refresh_InvClass()
  vsql = "SELECT TC_Class, TC_Desc From TL_Trans_Class, TL_Tab_Class Where TC_ID = " &
    Val(CbRequest.BoundText) & " and TC_Class = TC_Entry_Code and TC_Method = 0"
  Me.DatInvClass.RecordSource = vsql
  DatInvClass.Refresh
End Sub

Public Sub cmdAdd_Click()
  Set mycode = mydb.OpenRecordset("select max(TH_ID) from TL_Trans_H")
  VARMOD = "A"
  If IsNull(mycode.Fields(0)) Then
    VARCD = 1
  Else
    VARCD = mycode.Fields(0) + 1
  End If
  datPrimaryRS.Recordset.AddNew
  editmod Me
  txtfields(0).Text = VARCD
  txtfields(9).Text = Date + Time
  txtfields(12).Text = curuser
  txtfields(2).Text = ""
  txtfields(3).Text = ""
  txtfields(4).Text = ""
End Sub
```

```
Public Sub cmdDelete_Click()
 respvar = MsgBox("Are You Sure ?", vbYesNo + vbQuestion, "Delete Information")
 If respvar = vbYes Then
  With datPrimaryRS.Recordset
  mydb.Execute "delete from TL_Trans_D where TD_Id = " & txtfields(0).Text
  mydb.Execute "delete from TL_Test where TS_Trans_Id = " & Val(CbRequest.BoundText)
  'mydb.Execute "delete from TL_Trans_Class where TC_Id = " & Val(CbRequest.BoundText)
  .Delete
  .MoveNext
  If .EOF And .RecordCount > 0 Then
     .MoveLast
  Else
     DatSecondaryRS.Refresh
  End If
  browsemod Me, datPrimaryRS
  MsgBox "Record Deleted"
  End With
 End If
End Sub

Public Sub cmdfirst_Click()
 VARMOD = "B"
 datPrimaryRS.Recordset.MoveFirst
 VARMOD = "E"
End Sub

Private Sub CbRequest_Click(Area As Integer)
 If Trim(Me.CbRequest.BoundText) <> "" Then
  tmpreq = Val(CbRequest.BoundText)
  vsql = "select TH_ID, TH_Prod_Id, PH_Name, TH_Platform, TC_Desc, TH_Release from
TL_Trans_H, TL_Product_H, TL_Tab_Platform where TH_ID = " & Val(CbRequest.BoundText) & "
and TH_Prod_Id = PH_Id and TH_Platform = TC_Entry_Code"
  Set mycode = mydb.OpenRecordset(vsql)
  TmpProdId = mycode.Fields(1)
  TmpPlatform = mycode.Fields(3)
  If IsNull(mycode.Fields(2)) Then
   txtfields(2).Text = ""
  Else
   txtfields(2).Text = mycode.Fields(2)
  End If
  If IsNull(mycode.Fields(4)) Then
   txtfields(3).Text = ""
  Else
   txtfields(3).Text = mycode.Fields(4)
  End If
  If IsNull(mycode.Fields(5)) Then
   txtfields(4).Text = ""
  Else
   txtfields(4).Text = mycode.Fields(5)
  End If
  Refresh_InvClass
  Refresh_InvMethod
  Refresh_ClassTest
  Refresh_MethodTest

 End If
End Sub

Private Sub CbStatus_Click(Area As Integer)
```

```
 If VARMOD <> "B" Then editmod Me
End Sub


Public Sub cmdlast_Click()
 VARMOD = "B"
 datPrimaryRS.Recordset.MoveLast
 VARMOD = "E"
End Sub

Public Sub cmdlook_Click()
 curlook = "OomFTMaintDesign"
 OomLK01.Show 1
 If Trim(curlook) <> "OomFTMaintDesign" Then
  VARMOD = "B"
  datPrimaryRS.Recordset.FindFirst curlook
  VARMOD = "E"
 End If
End Sub

Public Sub cmdnext_Click()
 VARMOD = "B"
 datPrimaryRS.Recordset.MoveNext
 If datPrimaryRS.Recordset.EOF Then datPrimaryRS.Recordset.MoveLast
 VARMOD = "E"
End Sub

Public Sub cmdprevious_Click()
 VARMOD = "B"
 datPrimaryRS.Recordset.MovePrevious
 If datPrimaryRS.Recordset.BOF Then datPrimaryRS.Recordset.MoveFirst
 VARMOD = "E"
End Sub

Public Sub cmdRefresh_Click()
 datPrimaryRS.UpdateControls
 If VARMOD = "A" Then
  datPrimaryRS.Refresh
 End If
 Datstatus.Refresh
 VARMOD = "E"
 browsemod Me, datPrimaryRS
End Sub

Public Sub cmdUpdate_Click()
 Dim VarOldStatus As Long
'On Error GoTo err_upd

 If Trim(Me.CbRequest.BoundText) = "" Or IsNull(CbRequest.BoundText) Then
    MsgBox " Empty Input..."
    CbRequest.SetFocus
    Exit Sub
 End If

 txtfields(12).Text = curuser
 txtfields(9).Text = Date + Time

 VarOldStatus = IIf(IsNull(datPrimaryRS.Recordset("th_status")), 0,
datPrimaryRS.Recordset("th_status"))
 'mywork.BeginTrans
```

```
 If VARMOD = "A" Then
  Me.datPrimaryRS.Recordset.Fields("TH_Trans_Type") = "MD"
  Me.datPrimaryRS.Recordset.Fields("TH_User_Id") = CurCode
  Me.datPrimaryRS.Recordset.Fields("TH_Proposal_Id") = Val(CbRequest.BoundText)
  Me.datPrimaryRS.Recordset.Fields("TH_Prod_Id") = TmpProdId
  Me.datPrimaryRS.Recordset.Fields("TH_Platform") = TmpPlatform
  Me.datPrimaryRS.Recordset.Fields("TH_Release") = txtfields(4).Text
  'Me.datPrimaryRS.Recordset.Fields("TH_Int_Test") = IIf(ChkIntegrity.Value = 1, "X", " ")
  'Me.datPrimaryRS.Recordset.Fields("TH_Sys_Test") = IIf(ChkSystem.Value = 1, "X", " ")
 End If
 VARMOD = "U"
 datPrimaryRS.UpdateRecord
' mywork.CommitTrans
 datPrimaryRS.Recordset.Bookmark = datPrimaryRS.Recordset.LastModified

 If VarOldStatus <> Val(Me.cbstatus.BoundText) Then
  Set mytempcode = mydb.OpenRecordset("select max(TD_seq) from TL_Trans_D")
  If IsNull(mytempcode.Fields(0)) Then
   TEMPVARCD = 1
  Else
   TEMPVARCD = mytempcode.Fields(0) + 1
  End If

  vsql = "insert into TL_Trans_D
(TD_seq,TD_Id,TD_Status_Date,TD_From_Status,TD_to_Status,TD_User_Id) values(" &
TEMPVARCD & "," & txtfields(0).Text & ",'" & txtfields(9).Text & "'," & VarOldStatus & "," &
cbstatus.BoundText & ",'" & curuser & "')"
  mydb.Execute vsql
 End If

 MsgBox "Record Committed"
 VARMOD = "B"
 datPrimaryRS.Recordset.MoveLast

 Exit Sub

err_upd:
 MsgBox Error
 MsgBox "All The Transaction will be removed du to an internal error"
 VARMOD = "B"
 'mywork.Rollback
 If datPrimaryRS.Recordset.AbsolutePosition <> -1 And VARMOD <> "U" Then
 ' mydb.Execute "delete from tmp_skills"
 ' mydb.Execute "insert into tmp_skills select fop_skills.tal_code
,fop_skills.[Ski_Code],fop_skills.[Ski_Cost],fop_skills.[Ski_curcd],fop_skills.[Ski_user],fop_skills.[Sk
i_Dtm] , fop_tab_skill.[cod_desc] from [Fop_Skills],[fop_tab_Skill] where fop_skills.[Tal_Code]=" &
datPrimaryRS.Recordset.Fields("tal_code") & " and fop_skills.[ski_code] =  [cod_code]"
 End If
 Exit Sub
End Sub

Public Sub cmdClose_Click()
 Screen.MousePointer = vbDefault
 Unload Me
End Sub


Private Sub CmdInsMethod_Click()
 If Trim(Me.CbInvMethod.BoundText) = "" Or IsNull(CbInvMethod.BoundText) Then
    MsgBox " Empty Input..."
```

```
    CbInvMethod.SetFocus
    Exit Sub
  End If

  Refresh_TransMethod
  varcriteria = "TC_Method = " & Val(CbInvMethod.BoundText)
  DatTransMethod.Recordset.FindFirst (varcriteria)

  Set mycode = mydb.OpenRecordset("select max(TC_Seq) from TL_Trans_Class")
  If IsNull(mycode.Fields(0)) Then
    VARCD = 1
  Else
    VARCD = mycode.Fields(0) + 1
  End If

  If DatTransMethod.Recordset.NoMatch Then
    'DatSecondaryRS.Recordset.AddNew
    'DatSecondaryRS.Recordset.Fields("TC_ID") = Val(CbRequest.BoundText)
    'DatSecondaryRS.Recordset.Fields("TC_Class") = DatTransClass.Recordset.Fields("TC_Class")
    'DatSecondaryRS.Recordset.Update
    vsql = "insert into TL_Trans_Class (TC_seq,TC_Id,TC_Class,TC_Method) values(" & VARCD &
"," & Val(CbRequest.BoundText) & "," & DatTransClass.Recordset.Fields("TC_class") & "," &
Val(CbInvMethod.BoundText) & ")"
    mydb.Execute vsql
    DatSecondaryRS.Refresh
  Else
    MsgBox "Method already added" + vbCritical
    Exit Sub
  End If

  If CmdRmvMethod.Enabled = False Then CmdRmvMethod.Enabled = True
  CbInvMethod.Text = ""
  CbInvMethod.SetFocus
  Refresh_TransMethod
End Sub

Private Sub cmdPrint_Click()
  Rep.ReportFileName = "RptChgEva.rpt"
  Rep.SelectionFormula = "{TL_Trans_H.TH_Id} =" & txtfields(0)
  Rep.Destination = 0 '(0 preciew 1 printer)
  Rep.Action = 1

End Sub


Private Sub datPrimaryRS_Error(DataErr As Integer, Response As Integer)
  'This is where you would put error handling code
  'If you want to ignore errors, comment out the next line
  'If you want to trap them, add code here to handle them
  MsgBox "Data error event hit err:" & Error$(DataErr)
  Response = 0  'Throw away the error
End Sub

Private Sub datPrimaryRS_Reposition()
  Screen.MousePointer = vbDefault
 ' On Error Resume Next
  'This will synch the grid with the Master recordset

  'This will display the current record position for dynasets and snapshots
  'datPrimaryRS.Caption = "Record: " & (datPrimaryRS.Recordset.AbsolutePosition + 1)
```

```
 If VARMOD = "U" Then Exit Sub
 enabbutton Me, "cmddelete"
 OomMen011.Toolbar1.Buttons(12).Enabled = True

 Select Case datPrimaryRS.Recordset.AbsolutePosition
Case -1
   OomMen011.Toolbar1.Buttons(4).Enabled = False
   OomMen011.Toolbar1.Buttons(3).Enabled = False
   OomMen011.Toolbar1.Buttons(1).Enabled = False
   OomMen011.Toolbar1.Buttons(2).Enabled = False
   OomMen011.Toolbar1.Buttons(7).Enabled = False
   OomMen011.Toolbar1.Buttons(12).Enabled = False

   'cmdlast.Enabled = False
   'cmdnext.Enabled = False
   'cmdfirst.Enabled = False
   'cmdprevious.Enabled = False
   'cmdDelete.Enabled = False
   'cmdlook.Enabled = False
Case datPrimaryRS.Recordset.RecordCount - 1
   OomMen011.Toolbar1.Buttons(4).Enabled = False
   OomMen011.Toolbar1.Buttons(3).Enabled = False
   OomMen011.Toolbar1.Buttons(2).Enabled = True
   OomMen011.Toolbar1.Buttons(1).Enabled = True

   'cmdlast.Enabled = False
   'cmdnext.Enabled = False
   'cmdfirst.Enabled = True
   'cmdprevious.Enabled = True
Case 0
   OomMen011.Toolbar1.Buttons(4).Enabled = True
   OomMen011.Toolbar1.Buttons(3).Enabled = True
   OomMen011.Toolbar1.Buttons(1).Enabled = False
   OomMen011.Toolbar1.Buttons(2).Enabled = False

   'cmdlast.Enabled = True
   'cmdnext.Enabled = True
   'cmdfirst.Enabled = False
   'cmdprevious.Enabled = False
Case Else
   'cmdlast.Enabled = True
   'cmdnext.Enabled = True
   'cmdfirst.Enabled = True
   'cmdprevious.Enabled = True

   OomMen011.Toolbar1.Buttons(4).Enabled = True
   OomMen011.Toolbar1.Buttons(3).Enabled = True
   OomMen011.Toolbar1.Buttons(2).Enabled = True
   OomMen011.Toolbar1.Buttons(1).Enabled = True
End Select

 vsql = "SELECT * FROM TL_Trans_H where TH_Trans_Type = 'MD'"
 Me.datPrimaryRS.RecordSource = vsql
 'datPrimaryRS.Refresh

 If datPrimaryRS.Recordset.AbsolutePosition <> -1 And VARMOD <> "U" Then
   vsql = ""
   txtfields(0) = datPrimaryRS.Recordset("tH_ID")
```

```
   vsql = "SELECT TL_Trans_D.TD_Status_Date, TL_Tab_Status.TC_Desc,
TL_Tab_Status_1.TC_Desc, TL_Trans_D.TD_User_Id FROM TL_Trans_D, TL_Tab_Status AS
tl_tab_status, TL_Tab_Status AS tl_tab_status_1 WHERE TD_ID = " &
datPrimaryRS.Recordset("TH_ID") & " and TL_Trans_D.TD_From_Status =
TL_Tab_Status.TC_Entry_Code and TL_Trans_D.TD_to_Status =
TL_Tab_Status_1.TC_Entry_Code"
   Me.Datstatus.RecordSource = vsql
  Else
   Me.Datstatus.RecordSource = "SELECT TL_Trans_D.TD_Status_Date, TL_Tab_Status.TC_Desc,
TL_Tab_Status_1.TC_Desc, TL_Trans_D.TD_User_Id FROM TL_Trans_D, TL_Tab_Status AS
tl_tab_status, TL_Tab_Status AS tl_tab_status_1 WHERE TL_Trans_D.TD_From_Status =
TL_Tab_Status.TC_Entry_Code  and tL_Trans_D.TD_to_Status = TL_Tab_Status_1.TC_Entry_Code
AND TD_ID = 0 "
   txtfields(0) = ""
  End If
  Datstatus.Refresh

End Sub

Private Sub datPrimaryRS_Validate(Action As Integer, Save As Integer)
  'This is where you put validation code
  'This event gets called when the following actions occur
  Select Case Action
    Case vbDataActionMoveFirst
    Case vbDataActionMovePrevious
    Case vbDataActionMoveNext
    Case vbDataActionMoveLast
    Case vbDataActionAddNew
    Case vbDataActionUpdate
     browsemod Me, datPrimaryRS
    Case vbDataActionDelete
    Case vbDataActionFind
    Case vbDataActionBookmark
     'browsemod Me, datPrimaryRS
    Case vbDataActionClose
     Screen.MousePointer = vbDefault
  End Select
  Screen.MousePointer = vbHourglass
End Sub

Private Sub datSecondaryRS_Reposition()
'If datSecondaryRS.Recordset.AbsolutePosition <> -1 Then
'   CbSkills.BoundText = datSecondaryRS.Recordset.Fields("ski_code")
'   txtfields(11).Text = datSecondaryRS.Recordset.Fields("ski_cost")
'End If
End Sub



Private Sub Form_Activate()
  If curlook <> "First" Then Exit Sub
  If Not datPrimaryRS.Recordset.EOF Then datPrimaryRS.Recordset.MoveLast
  txtfields(12).Text = curuser
  curlook = ""
  VARMOD = "E"
End Sub

Private Sub Form_Load()
  'Create the grid's recordset
  'datPrimaryRS.Refresh
```

```
  curlook = "First"
  VARMOD = "B"
End Sub

Private Sub Form_Unload(Cancel As Integer)
  Screen.MousePointer = vbDefault
End Sub

Private Sub RichTextBox1_KeyPress(KeyAscii As Integer)
 editmod Me
End Sub

Private Sub RichTextBox2_KeyPress(KeyAscii As Integer)
 editmod Me
End Sub

Private Sub RichTextBox3_KeyPress(KeyAscii As Integer)
 editmod Me
End Sub

Private Sub Text1_Change()
'editmod Me
End Sub

Private Sub Text1_GotFocus()
editmod Me
End Sub

Private Sub txtfields_KeyPress(Index As Integer, KeyAscii As Integer)
If KeyAscii = 13 Then
If Index = 11 Then
   'Index = 0
   SSTab1.Tab = 1
   RichTextBox1.SetFocus
Else
If Index = 3 Then
   CbCountry.SetFocus
   ElseIf Index <= 6 Then

   txtfields(Index + 1).SetFocus
   Else
   CbFields.SetFocus
   End If
End If
Else
   editmod Me
End If


End Sub
```