INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Université d'Ottawa · University of Ottawa

Design and Implementation
Of An Integrated Training and Decision Support System
For the Activated Sludge Process

By

© Phạm Vũ Anh

A Thesis submitted to the School of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of
MASTER of SCIENCE
In Systems Science
Faculty of Administration
University of Ottawa

December 1997

ABSTRACT

In North America, the activated sludge process is commonly used in wastewater treatment. However, there are many aspects of the process that elude researchers, in particular, those that will ensure a successful control and optimization of the process. Because of the biological nature of the activated sludge process, its characteristics are highly dependent on environmental factors and hence can be very difficult to control and manage successfully. A very important factor contributing to an optimized activated sludge plant is the experience of its operators with the activated sludge process. This is achieved through the use of a variety of tools including traditional classroom training, computer based training, expert systems and simulation tools. Some of the tools aim to help the new operators acquire this experience quicker while others aim to enhance the experience of seasoned operators. This project presents a premise that an integration of these tools will be an effective and innovative way to help the operators achieve their goals. Hence, a design of an Integrated Training and Decision Support System (IT-DSS) has been proposed, and parts of the prototype system have been built. Preliminary results showed that such an integrated solution is feasible and realisable although a fully functional system has yet to be realised due to the scope limit. The results of this project provided solid and valuable starting points for the completion of such a step, and they also further support the value of the integration approach that has been put forth.

ACKNOWLEDGMENT

The ancient has said that every journey begins with a first step. I would like to thank my supervisor, Dr. Gilles Patry for showing me that first step and many more after that. His intuition, guidance and support have helped me greatly to meet my challenges successfully. I am also grateful that Dr. Patry always tried his best to be available, despite his busy schedule as the Dean of the Faculty of Engineering, when I began my work and later as the Vice Rector of Research. Next, I would like to express my gratitude to Dr. Ahmed Karmouch for his help in dealing with the rapidly changing information technology issues. Without his expert advice, I could have to learn many lessons the hard way, one time too often. I would like to also thank Dr. Dan Lane for his excellent supports and counsels that he provided to myself, as to all other students in the System Science Program as its former Director. And last but not least, I am grateful to my wife Thắm for her seemingly endless supports to keep me motivated throughout my work, and then to act as my personal editor to keep my writing from becoming so tiresome that even I would fall asleep reading it.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

INTRODUCTION

1.1 Waste water treatment and the activated sludge process

The treatment of wastewater prior to its discharge into receiving water bodies such as lakes, rivers or oceans is a vital requirement to protect the environment and to support urban and industrial activities. Although wastewater management is a broad subject involving complex socio-political issues in addition to advanced technical challenge, the most visible and possibly the most critical component is the wastewater treatment plant (WWTP).

1.1.1 Overview of the wastewater treatment process

Wastewater entering a treatment plant comes from a collection system and is called the influent, and the treated water discharged from the plant to the receiving water body is called the effluent. The influent stream typically goes through three main stages involving pre-treatment, primary treatment and secondary treatment, where waste characteristics such as biochemical oxygen demand (BOD) and suspended solids (SS) are reduced to an acceptable limit prior to being discharged into a receiving water body. The BOD and SS limits are generally stipulated by the regulator in the operation license of the plant. The purpose of the plant is to meet and/or exceed the effluent quality specified by the regulator.

A flow diagram of a typical wastewater treatment process is shown in **Figure 1 - 1.** The pre-treatment stage generally consists of four processes: screening, grit removal, pre-aeration and flow measurement. Screening removes roots, rags and other large debris; grit removal removes sand and gravel; pre-aeration helps remove oil and reduces odour problems. Flow measurement records are generally kept at this stage in order to track the flow entering the plant. Primary treatment follows where the settleable and floatable materials are removed and transferred to the solid handling process. The wastewater now contains only dissolved and non-settleable waste as it proceeds to the secondary treatment. Secondary treatment reduces the remaining waste to acceptable concentration *via* a biological process such as activated sludge or trickling filter or *via* chemical or physical processes as a combination thereof. The type of the secondary process used also gives the plant its classification, as a biological plant is usually identified as an activated sludge plant or a trickling filter plant to the people in the field.

1.1.2 Overview of the activated sludge process

The activated sludge process refers commonly to a class of biological waste reduction processes where the wastewater is transformed and in the presence of molecular oxygen (also known as aerobic digestion). This process is widely used in North America and continues to be the subject of active research. As with other issues in wastewater treatment, the dominant problems facing researchers have shifted from those of design and construction to those of plant operation (Andrews,

1993). The goal is to accumulate additional knowledge to improve process stability, performance and in the most economical way.

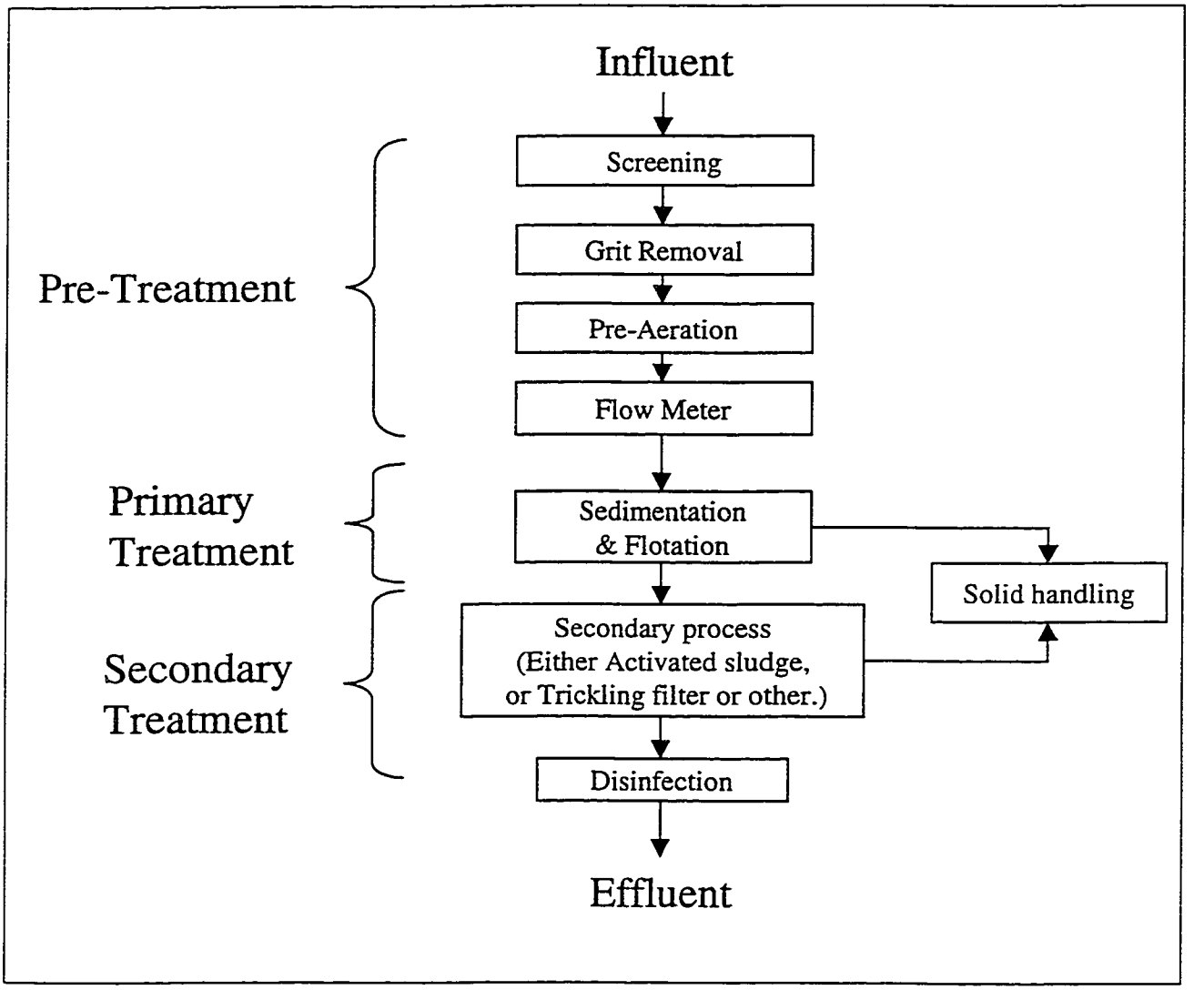


Figure 1 - 1 Flow diagram of a typical wastewater treatment plant

The activated sludge process uses the metabolic reactions of microorganisms to produce a high quality effluent by oxidizing the organic matter and thus reducing the oxygen demand of the effluent. In the basic activated sludge process (**Figure 1 - 2**), the wastewater enters an aerated tank where the microbial mass also referred to as biomass (RAS) is brought into contact with the primary effluent rich in dissolve

organic matter. Aeration is achieved by mechanical mixing, air diffusion or via pure oxygen diffusion. Under controlled conditions, the microorganisms metabolize the organic material into carbon dioxide and other end products and new biomass. A measure of biomass concentration in the aeration basin is represented by mixed liquor suspended solids (MLSS) or mixed liquor volatile suspended solids (MLVSS). The contents of the aeration basin are referred to as mixed liquor.

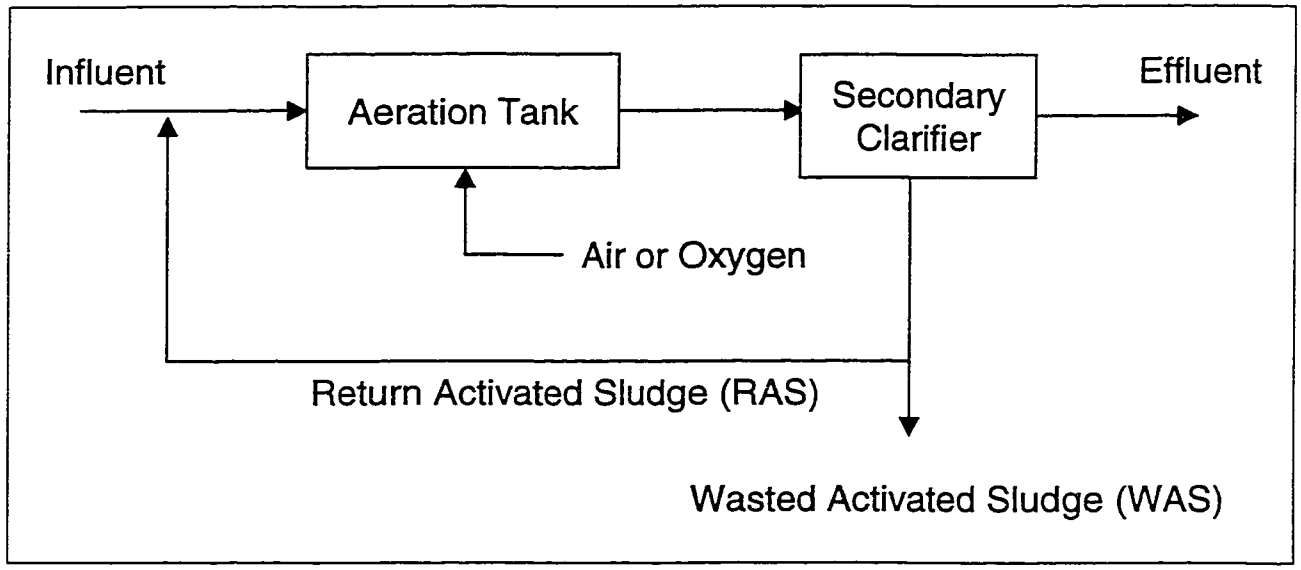


Figure 1 - 2 A typical activated sludge process

After the mixed liquor is discharged from the aeration tank, a clarifier (also known as settling tank or sedimentation tank) separates the suspended solids (SS) from the treated wastewater. To maintain high biomass concentration in the aeration basin for enhanced biological activity, the concentrated biological solids are recycled. This recycle stream is known as the return activated sludge (RAS) and is one of the process' control parameters. Microorganisms are continuously produced and must be kept under control, so the excess sludge is withdrawn from the clarifier

to be disposed of by the solid handling process. This stream is called waste activated sludge (WAS) and is also a control parameter.

Most activated sludge processes are used to remove carbonaceous biochemical oxygen demand (CBOD), but the process can also be designed to oxidize ammonia (nitrification process) or to remove phosphorous. The activated sludge process referred to in this thesis is based on CBOD removal.

1.1.3 Overview of support systems in a typical WWTP

Plant management is a subject that is beyond the scope of this thesis. However, it is necessary to briefly describe the different systems that exist and are operating cooperatively within a plant to provide conceptual clues as to where the work described in this thesis can be used. Most of the materials in this section is based on information published by the Water Pollution Control Federation (WPCF), in the series *Manual of Practice* (WPCF, 1990).

In a large plant, there are usually four cooperating systems: personnel management, process control, maintenance management and laboratory management system. The personnel management system includes a record system and other elements required to provide structure, direction, and control to the employees of the plant. The process control system (PCS) is the plant's most important management system because poor plant performance usually results from failure to apply known process control concepts. The PCS describes in writing how processes should be controlled by:

- Assigning responsibilities for individual processes,
- Presenting the methods for checking performance, and
- Explaining the procedures for changes necessary to produce the required results.

The PCS progresses through planning, implementation and management phases. Planning and implementation are normally associated with plant start-up and can be changed overtime with the plant's dynamics. However, it is in the management phase where the day-to-day actions of the planning phase are visible and carried out by plant's operators. This particular part of the PCS is also the focus of this thesis.

The process control system can not fulfill its objectives without direct support of the maintenance and laboratory management systems. The maintenance system is responsible for the plant logistics. It includes a preventive maintenance program, a maintenance record system, and a spare parts inventory system. Depending on the size of the plant and how modern the PCS system, it may also include an organization and staffing functions element, training, computerized data system, work orders planning and scheduling, and engineering support.

1.2 Overview of current application of information technology in plant activities

In the recent training materials for wastewater treatment plant operator (California State University, 1996), it was suggested that computer hardware and software, have now become affordable and easier to use, to be considered in improving plant operations and maintenance. In a WWTP, the most visible

application of computer technology is the Supervisory Control And Data Acquisition (SCADA) systems. SCADA systems collect, store and analyze information about all aspects of operation and maintenance. They transmit alarms when necessary and allow complete control of alarms, equipment and processes. SCADA systems are also capable of analyzing data. They can also provide operating, maintenance, regulatory and annual reports (California State University, 1996). These systems can also keep track of the performance history of equipment, prepare daily, weekly and monthly maintenance schedules, and help operators monitor the spare parts inventory system.

Other uses for computer technology are in record keeping (SCADA database management system), data analysis and reports creation (AllMax, 1996). It is noteworthy to point out that SCADA systems can only record and maintain data that can be acquired on line. Some examples of this type of data in the activated sludge process are flow rate, pH, temperature, pumping rates, power consumption, oxygen or airflow rate. Other vital information such as chemical oxygen demand (COD), BOD and various solids concentrations have to be tested off-line and recorded by the operator, stored in some media for later analysis. The applicable computer software regularly used for this task is a general-purpose spreadsheet program or specialized software packages such as those offered by AllMax Professional Services (AllMax, 1996).

Noticeably absent from the general operator training information is a discussion on computer assisted knowledge acquisition tools such as computer based expert systems (also known as knowledge based systems) and process simulators (Patry and Chapman, 1989). Expert systems can be useful in helping the operator to perform diagnostics, and process simulators can be used to understand and plan process control strategies. The discussion of the use of computer in assisting the training and certification of operators is also missing, although multimedia computer based training (CBT) packages are now becoming available (Water Environment Federation (WEF), 1996; Lewis & Zimmerman Associates (LZA), 1996).

1.3 Motivation for an Integrated Training and Decision Support System (IT-DSS) for the activated sludge process

The activated sludge process depends on active microorganisms for a successful operation and is intrinsically complex. Because this is a "living system", it is sensitive to environmental changes. Therefore, successful operation can only be achieved if the operator is able to recognize system changes and trends to make the proper decisions to successfully counteract potentially harmful changes (WPCF, MOP11, 1996). It has been also recognized that these decisions should reflect "judgment gained from understanding the biological and chemical principles at work and from past experiences with the system, rather than on rigidly applied general rules developed by others" (WPCF, MOP11, 1996). Unfortunately, this

knowledge and experience of the operator can be difficult to acquire. Furthermore, the unique nature of a particular WWTP will require that operators transferred from one plant to another adapt to changes in plant operation and performance.

Operators normally acquire knowledge through their own experience and training and by using simulation tools (Patry and Takács, 1994) and expert systems (Patry and Chapman, 1989). Operating tables, trend charts, and response curves generated by operators and researchers are generally available to plant personnel (Barnett and Andrews, 1990). In practice, fewer problems are encountered in plants where operators are well trained and experienced, or where the knowledge of experts is readily accessible. Thus, it is clearly advantageous to make the knowledge of experienced operators and experts, as well as new research information, readily available to less experienced operators for both training and operation support purposes (Barnett and Andrews, 1990; Sassen *et al.*, 1994; Boy and Mathe, 1993). One form of this knowledge is the archive of more than 14,000 pages of published articles, reports and case studies available from WEF (1996-1997) on searchable CD-ROM (WEF, 1996).

Research on activated sludge processes includes experimental studies for elucidation of important biological processes and controlling factors, as well as modeling based on first principles to test the validity of theory, and field studies for documentation of the knowledge used in actual operations (Patry and Chapman, 1989). These investigations have produced a large body of knowledge, which is

useful to activated sludge process operators in wastewater treatment plants. However, the efficient transfer of knowledge is often hindered by the individual characteristics of each process and by limited effectiveness of traditional knowledge transfer methods such as classroom training and operating manuals. Lessons are easily forgotten because operators may not see the immediate application of theoretical knowledge presented; operating manuals, even if well written, often require careful examination to identify possible solutions to complex problems and, thus, tend to lie unused (Krichten *et al.*, 1991).

Computer based training (CBT) packages provide a flexible learning environment, but do not address the gap between knowledge learned and actual situation where knowledge is applied. In the area of training, the system proposed in this thesis is designed to supplement regular computer based training (CBT) packages. The philosophy is to keep pertinent information as concise as possible and to provide an efficient way to access relevant information on an on-going basis.

With the advance of early expert system technologies, a subset of artificial intelligence research, several investigators have focused their efforts on the use of expert systems to address information accessibility problems (Barnett and Andrews, 1990; Barnett and Andrews, 1992; Ozgur *et al.* 1994). However, these earlier expert systems have all suffered from the limited amount of knowledge that can be encoded and the translation of this knowledge in a computer-based format. Kidd and Welbank (1984) conceded that the most difficult and time consuming and thus

expensive task is proper knowledge acquisition due to the lack of understanding of how an individual formulates, transmits and uses knowledge. This difficulty also limits the applicable range of the knowledge and restricts the incremental update of the knowledge base. Furthermore, some of the pioneering works can introduce a new communication barrier in communicating the encoded knowledge *via* primitive user interface such as text dialogue, where concepts are described as texts, leaving chances for misinterpretations.

This thesis proposes to maximize the usefulness of an expert system by limiting its application to defining and creating a core knowledge base that consists of relatively stable and common knowledge about the activated sludge process and keeping specific process details in a different system. This system records and manages problem cases that are specific to a process at a given plant. These problem cases can be queried and searched, and they represent a form of experience associated with a particular process. The actual definition and creation of a common core knowledge base is beyond the scope of this thesis. This work will concentrate on providing the access tools to this core knowledge base and the management of a repository of problem cases specific to each plant. The proposed access tools include a programmable interface to the expert system and multimedia user interface to help reduce the communication barrier in communicating the encoded knowledge.

From the computing perspective, a problem case is a special case of an electronic document, and the idea for creating a searchable repository for electronic

document is not new. This is the basis idea behind Electronic Document Management System (EDMS), and it has been used successfully in helping lawyers and physicians find similar cases studies in their line of work (Koucopoulos and Frappaolo, 1995).

Mathematical models for a number of activated processes are commercially available (Patry and Takács, 1994), and are being researched in academia. However, no one model can yet capture all the fundamental behaviour of an activated sludge process, a quest that is still the subject of active research. Although mathematical models could never truly emulate the real process, their use is valuable in testing control strategies and providing insight when the model results are compared with actual plant data.

As a rule rather than an exception, each model has a set of assumptions and conditions that define where it is applicable. To effectively use a model, one must understand the contextual information, a non-trivial task to most operators because of the academic language used in describing the information. What is potentially useful is a simple system to catalogue the models and properly guide their uses *via* descriptive interfaces. Researchers who wish to make the latest models available can do so by conforming to the requirements of this cataloguing system.

In summary, there are three major reasons that have motivated the design of an Integrated Training and Decision Support System (IT-DSS). The first one is integration. The cognitive challenges required to learn how to operate all of the

stand-alone systems can be overwhelming. By integrating the most useful elements of each system and presenting their usefulness to the operator in a logical manner, the potential for actual use of the system will increase. The next reason is focus. The operator can focus on what is relevant to the situation instead of being drowned in a sea of more or less relevant information. Finally the third reason is innovation. Because the available computer based tools (CBT, Expert Systems, simulation,...) have their own strengths, all could be further enhanced and improved through the use of an IT-DSS.

The task of a process control operator is to collect and analyze data for operational control purposes. It is around these tasks that the tools must be designed. A typical scenario would be that the operator sees a disturbing trend on a plotted chart and activates an expert system assisted diagnostic module, or searches through the repository of problem cases that have been logged by previous operators. In another scenario, the operator may wish to experiment with a control strategy and is unsure whether it would work. To try out the hypothesis, he/she makes use of the mathematical models that maybe able to suggest changes in operating conditions. Finally, another scenario would be that the operator consults the on-line documentation for some quick answers regarding some terms or certain conditions that he/she was not sure about.

All of the above scenarios represent typical "light weight" use of information, but particularly important ones. The Integrated Training and Decision Support

System proposed in this work will be designed to handle such "light weight" information demands. The goals of this thesis are to:

1. Evaluate the feasibility of designing the structure of the IT-DSS.
2. Evaluate whether the system is realisable.
3. Synthesise conceptual and specification design.
4. Evaluate the system design *via* a prototype implementation

Chapter 2

SYSTEM REQUIREMENTS

Several requirements can be synthesized and presented following the discussion of the previous chapter. These requirements form the basis for analysis, modeling and the design of the Integrated Training and Decision Support System (IT-DSS). Furthermore, they represent the desired features of the system that may be met in whole or in part by the successive prototypes of the IT-DS System, as development of the system progresses.

2.1 Information requirement of a Process Control System (PCS)

The raw material that is processed by a WWTP is wastewater and the product is acceptable clean water. Therefore, water composition in terms of chemical, physical and biological characteristics is crucial information that must be available to effectively control the activated sludge process and to evaluate its performance. Therefore, the first requirement of the IT-DSS is to be able to help the operator to implement any sampling and analysis plan of the Wastewater Treatment Plant (WWTP), collect data, analyze data and report information in a tabulated format or in a graphical format (i.e., charts). The system must also support established sampling practices and protocols, which specify the type of analysis to be carried out (e.g., COD, BOD, SS, NH_3, pH,...) and the frequency of

sample collection. It must also support data acquisition of the sampling process in term of identification, tracking and recording test results.

Planned actions, supporting rationales and other relevant information must be communicated to the operators, so that a Process Control System (PCS) can work effectively. Hence, the IT-DSS system must also provide ways of communicating such information. In a plant, this is normally achieved by updating operating manuals whenever there are changes in the PCS.

2.2 Additional requirements for IT-DSS

Aside from the above requirements, the IT-DSS must also provide features to support process diagnosis and to forecast process performance. This can be achieved by providing historical process data management, the use of expert systems and of process models and simulation tools.

2.2.1 Support for process historical data management

History data (problem cases) document the detection and definition of problems, as well as how the problems were diagnosed and what solution(s) was used or attempted. Sometimes, a wrong diagnosis could result in a sub-optimal control strategy and possibly severe plant upset. Records of these situations must also be logged by the system.

In a plant, these cases are normally recorded in logbooks and are supposed to be helpful in future process troubleshooting. However, these

informal logbooks are rarely well organized to facilitate the reuse of information, or are frequently ignored altogether. By having a computerized system to record, organize and facilitate the searching and browsing of logged problem cases, the benefit of keeping logbook information is enhanced.

2.2.2 Support for using expert system

It is equally important that the IT-DSS provide support for expert system use. As proposed previously, the knowledge on process operation and diagnostic is partitioned into a historical data portion (**Section 2.2.1**) and a core expert system portion that focuses on common and relatively well-known knowledge about activated sludge process. The IT-DSS will provide an interface and representation package so that this core knowledge base can be accessed with an expert system mechanism (*i.e.* using an expert system shell). This core knowledge is accessed as the needs arise as part of the overall diagnostic process.

2.2.3 Support for using process models

The IT-DSS system will be designed to support process models. This requires that the system will provide a means to select the appropriate model and use models of activated sludge processes that are stored and catalogued. The basic parameters could be drawn directly from information kept in the system for a particular plant (*e.g.*, physical parameters, state parameters...), while the operator (user) provides the run time parameters (*e.g.*, operational conditions). The results of the simulation run(s) will be produced in graphical or

tabulated formats, and there will be an option for comparison with current process information. The system must also provide supports to update the models, so that model builders can provide new models or change existing model behaviours.

2.2.4 Support for on-line documentation

The IT-DSS will provide on-line documentation with search capability. This on-line documentation would take advantage of multimedia features to enhance understandability. The documentation also provides information at the level of detail specified by the operator. The level could be extensive, or concise and brief, or with context sensitive expansion to provide optional extra meaning. This on-line documentation must be adequately designed to handle changes that are transparent to the user.

2.3 General system requirements

As a desirable feature, the IT-DSS should allow a plant layout to be created and managed visually. This means that plant diagram can be assembled from pre-built symbol lists; the properties of the process' unit-operation (e.g., an aeration tank) such as dimension and design limits are recorded at the moment of creation, and can be modified later by interacting with the representative symbol. The data management tasks should also be accessed by interacting with the visual representation of the process. This means that the operator enters the data and analyzes these data by first selecting the visual representation of the

process unit-operation. The data analysis task is then completed by using a graphical user interface (GUI).

The IT-DSS also provides services to store and retrieve data in and from databases; data such as water quality data, plant information, user information and documentation. Since there are several operators working on several shifts, the system must also support cooperative and shared data access and must be able to control access privileges. Finally, the operators should also not be concerned with computer-related details such as data storage, data access etc. The system administrator will be responsible with such details, the system administrator or some one with administrator privilege is the only one that can set-up the plant layout, change process property information, and assign user access rights.

Chapter 3

DESIGN METHODOLOGY AND ISSUES

Donald Firesmith (1993) defined software engineering as a discipline that would be concerned with "the establishment and application of current sound engineering concepts, principles, processes, methods, techniques, models, metrics, and environments including tools and languages." To be successful, software engineering must also be supported and enforced by policies, standards, procedures, guidelines, practices, evaluation and tools.

Software quality is a complex attribute. Ideally, quality software means software that is correct, efficient, extensible, flexible, maintainable, modifiable, portable, reliable, reusable, safe, testable, verifiable, and understandable (Firesmith, 1993).

Whether current software development practices are qualified as an engineering discipline is still a controversial and complex issue. This discussion is beyond the scope of this thesis. Nevertheless, a systematic approach to software development must be chosen, so that basic engineering principles can be implemented systematically. To this end, object-oriented software development (OOSD) approach and design experiences that are expressed as design patterns will be used. The relevant details of both OOSD and design patterns will be discussed in the following sections.

3.1 Object-oriented software development

The object-oriented software development (OOD) paradigm emerged in the 1980's and has been evolving as a collection of methodologies and guiding principles in building software systems. OOD attempts and to some degree has succeeded in dealing with the complex issues of large-scale software development and software evolution. There are numerous treatments on this paradigm that can be consulted (Fowler and Scott, 1997; Martin and Odell, 1996; Sigfried, 1996; Booch, 1994; Firesmith, 1993; Jacobson, 1992; Rumbaugh *et al.*, 1991; Coad and Yourdon, 1991).

3.1.1 Fundamental concepts

OOD builds on the concept of object, which could represent real world objects or abstract entities. An object communicates to the surrounding world *via* its publicly recognizable *states* and *behaviours*. The concept of object allows the designer to deal with complexity by reducing the semantic gap between the objects in the real world and those found in software system. Other fundamental concepts are *abstraction, polymorphism, inheritance* and *encapsulation*.

Abstraction is a process that helps humans deal with complexity. This process is often used by a person to extract relevant information from a real world object. Thus, unnecessary details of an object can be ignored to allow for better understanding of the object. Using abstraction, a software designer can focus on the desired characteristics of the system. OOD merely facilitates the use of abstraction,

not only in understanding the problem, but also in building a software system using the same abstraction. The concepts used in expressing abstraction is a *class* (a collection of similar objects), an *association* (relation between classes), *multiplicity* (quantity of objects that can participate in an association) and *type* (objects of the same type share the same interface that is exposed to the world).

Polymorphism means that the object will react to the world depending on what the world gives to it. In the real world a message is perceived differently because of the receiving party's internal states and thus is reacted to differently. Polymorphism is thus a construct that permits software to behave more like its real world counterpart, minimizing again the semantic gap between the behaviours of real world objects and those of software objects.

Inheritance is a special kind of association which expresses that one object is a specialization of another and thus acquires some characteristics from its more general form. From the modeling point of view, inheritance helps in understanding the world. From the software efficiency point of view, inheritance represents reuse of functionality that has been developed in a more general case.

Encapsulation is the act of hiding private details from the world. By disallowing uncontrolled access to an object's internals, encapsulation allows changes to take place without affecting normal operations of other objects; it enforces what can be done to an object. Encapsulation organizes data and the operations, which other users can perform on the data, into one unit.

Association sets up object connection. An association can be hierarchical such as *inheritance* and *aggregation* (something has something(s), something contains something(s)), or non-hierarchical (and is called *relationships*) such as *work for, come from*.... Some authors also differentiate the degree of *association* (strong or weak) by examining the mutual responsibility of the parties involved (*e.g.*, the association between the car and its engine is a strong association, since the notion car implies that it has an engine). *Association* can also be directional (Martin and Odell, 1996; Booch, 1994; Rumbaugh *et al.*, 1991).

3.1.2 Object-oriented development process

From the late 1980's until the early 1990's, there was a wave of object-oriented analysis and design (OOA&D) methods offering both a modeling language and a process to the designer (Martin and Odell, 1994-1996; Jacobson, 1992-1995; Booch 1994-1996; Rumbaugh *et al.*, 1991; Coad and Yourdon, 1991; Beck and Cunningham, 1989; Shlaer and Mellor, 1989). The modeling language is a system of notation (mainly graphical) that methods use to express design in the form of object and class diagrams. The process is a collection of suggested steps to follow to achieve a design. The equivalent of a software design diagrams is electrical circuit drawing that express the connections (relationship) among various components (classes and objects) of the systems. The various processes proposed are usually the results of the accumulated experience in software analysis and design of various authors of a particular OOD method.

3.1.3 Modeling perspectives

Martin Fowler (1997) distinguished three types of object models: conceptual model, specification model and implementation model, which should be understood under appropriate perspectives. The conceptual model is built to capture the semantics of the problem domain and is use to communicate ideas between the designer and the user. This model is developed freely, without worrying about how a system can be designed and implemented. Next, the specification model establishes the interfaces between classes of objects. This model may include classes that have implementation semantics; however, it should focus only on the responsibilities of classes to each other. Finally, the implementation model includes the details of how the interfaces will be supported.

Booch (1996) discussed the strategic and tactical decisions facing a software designer. The designer must make strategic decisions, which determine the overall nature of the software system. Strategic choices include deciding whether the system is distributed or stand-alone and choosing a common architecture such as client-server. Tactical decisions help to optimize performance and economics and are concerned with hardware/software platform or a particular algorithm. However, before making these decisions, the designer must do a high level analysis of the problem domain, to learn about current established architectures and similar issues. These high level recurring patterns have recently been formally captured, analyzed and presented in a language form called *pattern* (Coad *et al.*, 1997; Bushmann *et al.*, 1996).

3.1.4 Architecture-driven software development

Booch (1996) enumerated five major styles of software development project that he called focus or culture. The style can be calendar-driven, requirements-driven, document-driven, quality-driven, architecture-driven, or a mix of several styles. The style that he advocated is architecture-driven development and is the one used in this project (refer to Booch (1996) for a detailed discussion and comparison of all five styles).

Architecture-driven development is characterized by the focus on creating a framework, which satisfies all immediate requirements and is resilient enough to adapt to those requirements that are not yet known or well understood (Booch, 1996). Adaptable software architecture can meet near term goals by allowing incremental building of functionality as the design evolves, and long term goals by incorporating feedback from these incremental developments.

It is well known that a user's requirements change as new or different solutions are proposed, and that it is impossible to anticipate all potential uses (Davis, 1995). Therefore, as real requirements are discovered, the architecture-driven development strives to accommodate these changes by building and evolving the software framework. The design decision focuses on how existing requirements can be met while allowing enough degrees of freedom for future requirements. This is equivalent to knowing where changes can be made that will preserve system integrity and stability, so that new requirements can be met safely.

The new changes also have to be designed such that they will not interfere with future requirements or that they will add to the stability of existing software structure.

Booch (1996) stated that architecture-driven projects typically have the following processes:

- Specification of the system's desired behavior through a collection of scenarios;
- Creation and validation of an architecture that exploits the common patterns found in these scenarios; and
- Evolution of that architecture, making mid-course corrections as necessary to adapt to new requirements as they are uncovered.

He also outlined elements for a well structure object-oriented architecture and called these items infrastructure layers:

- Base operating system
- Networking
- Persistent object store
- Distributed object management
- Domain-independent framework
- Application environment
- Graphical user interface (GUI) desktop environment (or just user interface)
- Domain model

- Domain specific framework

Base operating system and networking serve to insulate the rest of the application from the detail of its hardware. The choice of operating system depends on how well its services match the application requirements. Some types of applications do not need networking services, yet some require more advanced features (*e.g.*, support for real-time stream).

Built on top of the operating system and networking layers are persistent object store and distributed object management layers. Persistent object store provides services that will allow objects to persist beyond the boundary of an execution environment. Distributed object management builds upon these services and those from the networking and operating system layers to provide services for the administration of distributed and mobile data in a network environment. Together, these two layers provide to the application an illusion that objects live permanently in a large, virtual address space, that will make the application simpler, particularly in geographically distributed situations.

The next level of the infrastructure consists of domain-independent frameworks. These frameworks are primitives that provide facilities such as collection classes to manage collections of objects, application classes that will provide services such as clipboard management and printing, and graphical user interface (GUI) classes to build user interface. They are domain-independent framework, application environment and GUI/desktop environment, respectively.

The abstractions that are located at the top most level are specific to the problem domain. They are the domain-specific framework and domain model(s). The domain model serves to capture all object classes, associations, multiplicity, constraints, and type that form the vocabulary of the problem domain. The domain-specific framework provides all the common collaboration of these classes of the problem domain; however, it may not exist initially, and its design is generally left until several applications have been built successfully.

3.2 Design patterns

> "Design object-oriented software is hard, and designing *reusable* object-oriented software is even harder"(Gamma *et al.*, 1995).

Although the above quote is for object-oriented software design, it is equally true for software design using whatever paradigm. A good design should be specific to the problem at hand and it should be general enough to address future problems and requirements. However, a reusable and flexible design is difficult if not impossible to achieve on the first trial. Good designers do not solve every problem from first principles. They learn from their own experience and that of others and apply solutions that are well-known and have proven to work well. In object-oriented software design, there are recurring patterns of classes and communicating objects in many object-oriented systems (Gamma *et al.*, 1995). These patterns can be reused or learned upon to solve other specific design problems, making object-oriented design more flexible, elegant and reusable.

3.2.1 What is a software design pattern?

Software design pattern took roots in a totally unrelated field: architecture, from the work of Christopher Alexander, an architect, on patterns in building and towns (Alexander, 1979). In its most basic concept, a design pattern is a language form, and its purpose is to communicate well known solutions to design problems. Alexander proposed a system of cataloguing patterns, which he called a pattern language that would help young architects to find proven solutions to recurring design problems.

Patterns have taken root in software only recently. Coad (1992) first noted the link between Alexandrian patterns and software architecture. Since 1993, patterns have entered software consciousness as the result of seminars, conference sessions, and journal publications. In 1995, Gamma *et al.* (1995) published a book on *Design Patterns* that has become a classic. In this work the authors presented the first comprehensive treatment of a set of software patterns and proposed a style to express patterns in software that bear their names. Many other publications have since appeared on software pattern: general patterns (Coplien and Schmidt, 1995; Vlissides *et al.*, 1996), object models (Coad, 1995; Coad, 1997; Fowler, 1997), software architecture (Bushmann *et al.*, 1996), and large-scale software system (Mowbray and Malveau, 1997). The form that the authors used to describe patterns may follow different styles (Alexander, Gamma *et al.* or author's own), but the purpose is the same.

3.2.2 Using design patterns

Using patterns is advantageous, but not problem free. To use patterns effectively, the designers have to develop skills to recognize pattern(s) in the problem domain, so that they can be taken advantage of. This cognitive level is not something that can be taught easily. Most authors recommend using the patterns as starting points and iterating until the "right" one is found. Once a candidate pattern (or candidates) is found, it will have to be refined into useful forms. The concept of pattern is still a young idea, and any experience with its actual use is also valuable. Therefore, the experience with using patterns in this work will also be discussed to uncover problems and realistic advantages that are useful for future works.

3.3 Selection of development process

Fowler (1997) remarked that people only use the modeling language of a particular method, and rarely followed its process entirely. Therefore, the modeling language is probably the most important part of the method, for clear communication purpose. Any process or a combination of processes could be used to arrive at a design. In an industrial setting, rigorous process selection, extraction, combination and application is necessary to produce deliverables (or artifacts) to support the software life cycle.

For this work, a lightweight process is used instead. This lightweight process focuses on establishing solid software architecture and is influenced much by the thinking of Booch (1994 and 1996) and Fowler (1996 and 1997). It consists of an

architectural style analysis and selection, a development of domain model(s), a development of specification model(s), and a development of prototype implementation model(s), using design and analysis patterns wherever possible. This approach allows the work to be complete within scope and with reasonable time, while providing a base starting point for rigorous development once the concepts are proven useful and feasible (*i.e.*, using user feedback on the first prototypes). Moreover, this scheme fits well into the iterative and recursive nature of modern software development and is recommended by several authors in the initial phases (Martin and Odell, 1996; Davis, 1995; Firesmith, 1993).

The unified modeling language (UML) (Rational, 1997) will be used to express the designs in this work. The UML is a notation language, which consists of a system of graphical symbols and a formal semantic definition for the meaning of these symbols. The graphical symbols allow the designer to represents classes, objects, relationships and other OOD concepts in an well-understood and unambiguous way. A complete treatment of the UML is beyond the scope of this work. However, the elements of UML that are used in this thesis are easily understood with the OOD concepts that have been introduced in this chapter and with detailed discussion of the diagrams as they are presented. Furthermore, **Appendix A** provides a brief tutorial of UML 1.0 notation used in this thesis.

The UML is in the process of becoming a standard for object-oriented modeling and is being ratified by the Object Management Group (OMG). The

notations used are based on a relatively stable version of the UML (version 1.0). By using standard notations, the designs will be easily understood by future investigators and can be used in automatic software tools that understand UML.

Chapter 4

SYSTEM DESIGN

The Integrated Training and Decision Support (IT-DS) System design consists of two iterative phases, independent of design methodology: a logical design analysis phase and a specification phase. The goal of these two phases of work is to obtain a well-structured software architecture. This chapter reports and discusses the results of the design process as object models. These models are necessary for the construction of the first version of the prototype elaborated in the next chapter and in future versions.

The activities in the logical design analysis phase are designed at collecting high level information about the problem domain. This information is used to construct conceptual models to help formulate the problems, so that they can be better understood. These models can also be used to communicate with the users to gather their feedback. Using techniques such as interview with users, literature survey and use-case[1] analysis, this phase will identify the components of the system and the boundary imposed by the environmental constraints that these components must operate within. The deliverables are the conceptual models of the problem domain and strategic decisions on the general system architecture.

---

[1] The term "use-case" was coined by Jacobson (1992) to denote a grouping of all the scenarios representing a case of system usage among participating human and software entities that he called "actors" (refer to Appendix A for more details). The awkward English usage was probably unintentional. However, the term has become well known among practitioner of Jacobson's methodology and is left intact in the UML.

Once the domain is sufficiently understood and strategic decisions have been made, the second phase of the design can begin. At the end of this phase, the characteristics of the system architecture and the behaviour of the components are specified. The specification results from activities in finding the design solution that would satisfy the constraints identified in phase one as well as the strategic decisions that have been made. The design solution consists in mapping the conceptual models into specification models and details on the required external behaviour of components. These behaviours specify how issues such as persistence, distributed object management, environment and user interface are resolved. The deliverables are the specification models, where the interfaces of domain classes as well as supporting classes (*i.e.*, persistence, distribution, user interface, and environment) are specified.

4.1 Logical design analysis and modelling

The main purpose of an IT-DS System is to assist the wastewater treatment plant operator in controlling the activated sludge. It does this by assisting the acquisition and storage of process data and automating their analyses. It also enhances and augments the operator's process knowledge *via* on-line multimedia support material. Furthermore, it provides the operator with a tool to manage process operating history (as case-logs) and assists in the access and use of expert system and process simulation tools.

Several areas of information technology application can be identified: data management and data analysis, computer-assisted training, application of artificial intelligence (expert system) and numerical modelling (simulation). A literature review yielded many works on each of these areas, but little was found on the integration of the applications in these areas or the effects on using some of these tools together. Integration can provide (a) higher efficiency through the sharing of software function, and (b) lower costs due the purchase and maintenance of fewer systems.

The published works have broadened the scope of many areas. These include the application of advanced data management (Cattell, 1994; Rao, 1994) and advanced computer-assisted training. For example, intelligent tutoring system and computer-assisted instruction (Frasson and Gauthier, 1990; André and Rist, 1990; Burns *et al.*, 1991; Goodman, 1991; Larkin and Chabay, 1992; Sukaviriya *et al.*, 1992). Furthermore, there were works on bringing expert system and simulation together (Gall and Patry, 1989) and on expert system and multimedia (Garrity and Sipior, 1994). However, the scope was either different (*e.g.*, expert system that controlled the simulation (Gall and Patry, 1989)) or too specialized (*e.g.*, knowledge representation and access with multimedia (Garrity and Sipior, 1994)).

This lack of prior knowledge has caused some initial difficulties as to how the problem posed by this work can be approached. To limit the scope, the requirements were prioritized followed by a classification of tasks. It was observed

that tools such as expert system, simulation and on-line documentation were of secondary importance to the tasks of monitoring, collecting and analyzing process data. Therefore, the focus of the system was centered on the support for decision making which included data management as a major role and the use of expert system and simulation as supplementary roles. Training has also taken a support role in this new perspective. Its focus is now on helping the experienced operators to recall important information about the process and on guiding the inexperienced ones on the right path with annotated short lessons. In this way, the training feature complements the existing products on the market (*e.g.* WEF, 1996) and fulfills its support role.

Now that the focus of the IT-DS System has been established, the constraints of the environment, in which the system must work and must be designed for, will be identified. The first constraint is the ***affordability*** of a system. Although affordability is a relative concept, the overall cost of the system must be reasonable. This implies that the solution must involve as much off-the-shelve components as possible. The second constraint is ***reliability***. To satisfy this constraint, the system is required to be fail-safe, fault-tolerant and accountable. Reliability is important because process data are not only vital for operation but also important from a legal perspective. The system would not be trusted if it can not offer assurance that data is safe and that its analysis of the data is correct. Reliability also extends to the correctness of other data such as document, expert system knowledge and

simulation module. The third constraint is *time*. There are two time categories: time of system completion and system response time. For the first time constraint category, the system must be implemented within a reasonable time period. For the second time constraint category, it means that the design and choice for implementation technology must be made so that the system overall execution speed must be acceptable.

4.1.1 Overall system model

There are two human actors who will be interacting with the system: the operator and the system administrator. Both are users of the system. The operator is interested only in water records, case-logs, documents, simulations and knowledge base. The operator has read access to all of these entities, but he/she can only update water records and case-logs where permitted. On the other hand, the system administrator is responsible for the management of all these objects. To examine these scenarios, and eventually arrive at the specification phase, two types of information are used: object conceptual models and the results of use-case analysis.

Conceptual models help in identifying and understanding the problem domain, while use-case describes scenarios of interaction between user and system or between system components with one another. Use-case is formalized and extended by Ivar Jacobson (Jacobson, 1992) to help discover and understand the system requirements. Jacobson (1992) also uses the term *actor* to denote an entity

that participates in a use-case. Rumbaugh (Rumbaugh *et al.*, 1991; Rumbaugh, 1996) advocates using use-case to discover class operations (methods). These operations are the behaviours of a class towards itself and the outside world. They represent the responsibilities of a given class (Wirfs-Brock *et al.*, 1990). The use-cases involving the operator are illustrated **Figure 4 - 1**, while those involving the administrator are shown in **Figure 4 - 2**.

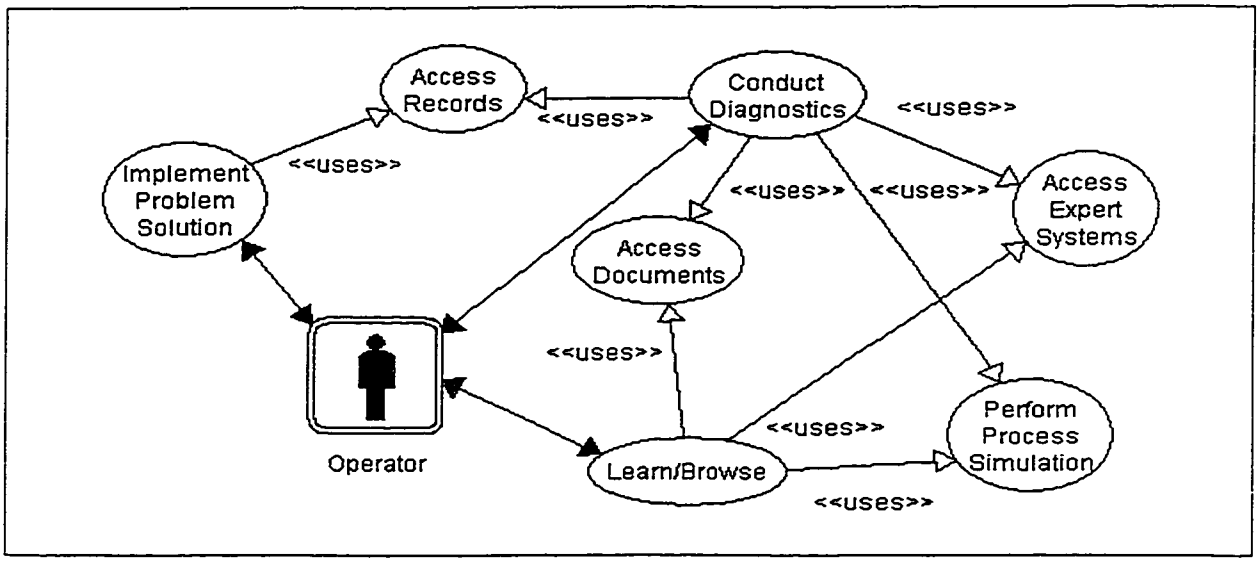


Figure 4 - 1 Use-cases involving the operator, where use-cases are represented by circles.

As illustrated in **Figure 4 - 1** the system needs to manipulate the object models of water record, case-log, meta-data and document, as well as those of several active objects such as numerical simulation model and expert system. The common use-cases for these classes of object are their creation and persistence, searching and retrieval, modification, displaying and cancelling. The special needs with water records and case-log are analysis and report. Those of active objects are

initialization (including dialogue with user if need be), activation and collection of results. Creation is unique to each class of object and so is its display. However, the processes of storing, retrieving, searching, modifying and canceling are similar for all classes. Hence, common system classes can be used to encapsulate these services so that all data objects can share them.

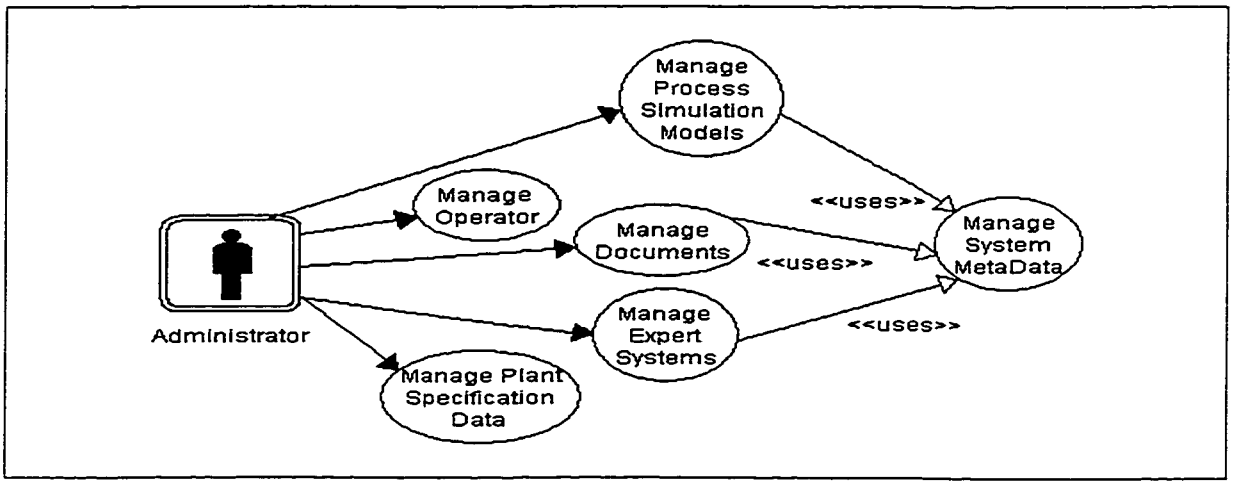


Figure 4 - 2 Use-cases involving the system administrator.

4.1.2 IT-DS System architecture model

At this point, a strategic decision can be made regarding the overall architecture of the IT-DS System. The system design should be based on a client-server (CS) architecture model. This model decentralizes services by partitioning them among many servers that many clients can access (Sommerville, 1995). Choosing a CS architectural model also benefits from proven strategies and reusable components that would help in meeting the reliability and time constraints. Furthermore, a CS architecture model facilitates load distribution so that operations that are computationally expensive such as simulation and expert system processing

can reside on powerful servers. With this arrangement, a simpler client can be developed to run on a less powerful and cheaper computer, thus reducing cost. This architecture is illustrated in **Figure 4 - 3.**

The common services and the application logic are encapsulated in the application server, while the administrator and operator interact with the system *via* a client package that provides graphical user interface (GUI) and a local cache. The GUI package is responsible for providing all of the required interaction functionalities. Whereas the local cache keeps frequently accessed data to speed data access and optimize server usage. Other functionalities reside on appropriate servers as shown. These servers can run on the same physical machine or distributed across machines.

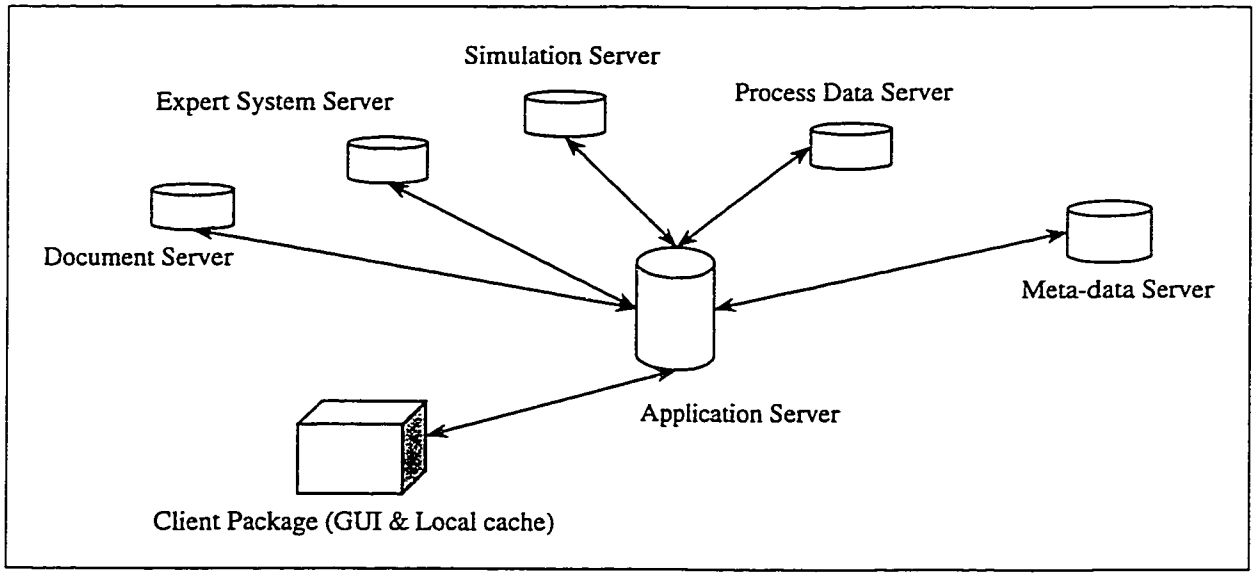


Figure 4 - 3 IT-DSS System architecture.

The domain object consists of the abstractions of interest in the IT-DS System.

These abstractions are manipulated by the servers that communicate with the application server and represent process data, document, expert systems modules, simulation models, and objects that have physical meaning in a plant environment such as unit-operation, receiving water and others. The Unified Modeling Language (UML) uses the concept of a *package* to tie together related classes and objects. Using the UML, the architecture model in **Figure 4 - 3** can be expressed as packages. These are the **domain object package**, the **user interface package**, the **persistence package**, the **distributed object management package** and the **environment services package**. The operating system and hardware layers are assumed to be implicit components of the system that already exist and are accessible *via* development tools such as programming languages and software libraries. The dependency of the various packages is shown in **Figure 4 - 4**. For example, domain objects depend on persistence, user interface and distributed object management, and these packages in turn depend on environment services. The conceptual models of these abstractions will be discussed in the following sections.

4.2 Domain object conceptual models

In this section, the conceptual models of the data and the components of interest to the IT-DS System are constructed and discussed. The actual implementation details are abstracted away to reveal only the logical relationship among the entities (classes) because the goal is to understand and to accurately find a representation of the real world entities.

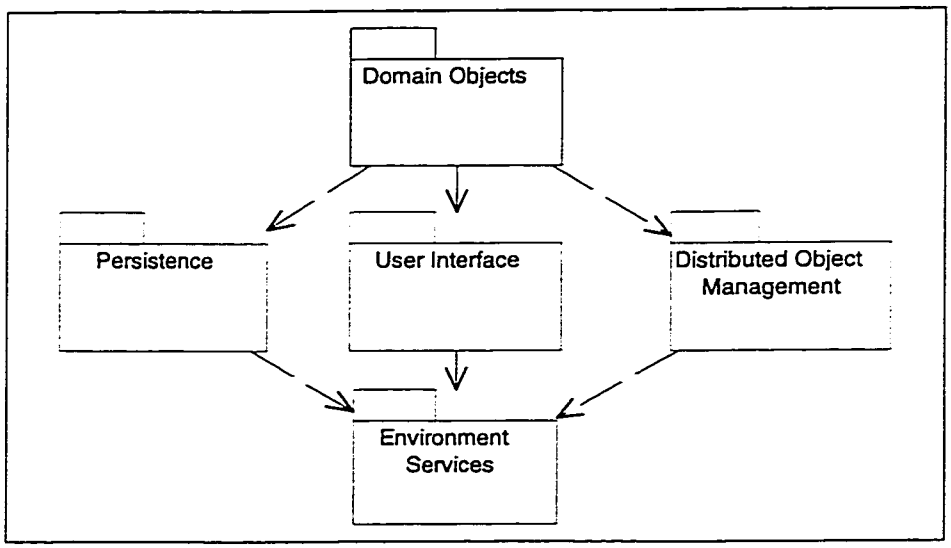


Figure 4 - 4 IT-DS System packages.

4.2.1 Overall conceptual model

From the conceptual point of view, the IT-DS System manipulates six classes of object: document, process data, meta-information (*i.e.*, meta-data), simulation model, expert system and history data (*i.e.*, case-log). Each will have its own conceptual model that describes in more detail the structure and dynamics that are considered internal from the overall conceptual view. For now, the relation among these six classes of object is of interest. The six classes can be further partitioned into two categories: document and meta-information into one category, and process data, simulation, expert system and history data into the other. This partition helps the understanding of the information that is manipulated.

Process data, simulation models, expert systems and history data can exist in the system without document and meta-information, although they would appear

totally cryptic to the uninitiated. For this reason, they are said to be dependent on document and meta-information to be fully useful. There exists a one to one association between an instance of document, process data, simulation model, expert system or history data and an instance of meta-information. The same one to one association exists between an instance of document and that of process data, simulation model, expert system, or history data.

For the operator, these relationships would result in optional documentation for the data set collected and searching and browsing are faster with the use of meta-information. However, these one to one connections would create a high dependency on document and meta-information. Therefore, one additional class called *descriptor* is introduced. Each instance of process data, simulation model, expert system, or history data has one instance of the descriptor class. Each instance of the descriptor class has connections to document and/or meta-information. This overall conceptual model is shown in **Figure 4 - 5.**

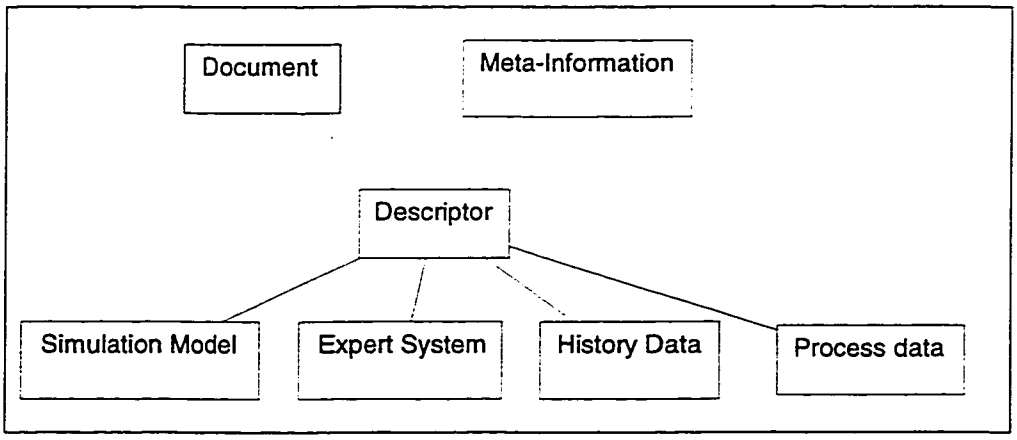


Figure 4 - 5 Overall conceptual model of manipulated classes in IT-DS System.

One of the requirements of the IT-DS System is inter-operability among simulation model, expert system, history data and process data. A simulation model may want to perform comparison between actual and simulated data. An expert system may need the process data as input parameters used in the inference process, while the history data package may need to keep a reference to the process data. Another requirement is that the physical parameters of the unit-operation or unit-process such tank size and flow rate must be automatically supplied to the components that use them. Components that may need to know these parameters include simulation model, expert system, history data and process data.

The advantage of automating the information exchange is that the user does not need to manually change the simulation model parameters, certain default information used in recording process data, or history data. In this regard, the model shown in **Figure 4 - 5** is incomplete. One possible improvement to the model is adding another class to represent these physical parameters. Another choice is breaking the process data into subclasses of physical process data and water process data.

In a plant setting, process data are sometimes used to mean everything about the process including water quality characteristics and physical parameters such as tank size, flow rate, time in service and so on. One possible choice of modeling the physical parameters is to make them subclasses of a generic process data base class, and using subclasses seems to fit this meaning of process data in a plant. However,

introducing a different class to represent fixed parameters of the process is more advantageous due to the clear separation of concept that would be the result. Therefore, the process data class is eliminated and is replaced by two new classes: process operation data and process specification data.

The process operation data (POD) represent observations and measurements collected about water samples, daily flow rate and, in short, entities that change from day to day. On the other hand, the process specification data (PSD) represent information about limits of operation of process and equipment, which would be changing only with equipment replacement or renovation. The new model is shown in **Figure 4 - 6.** In the following sections, each of these class models will be described.

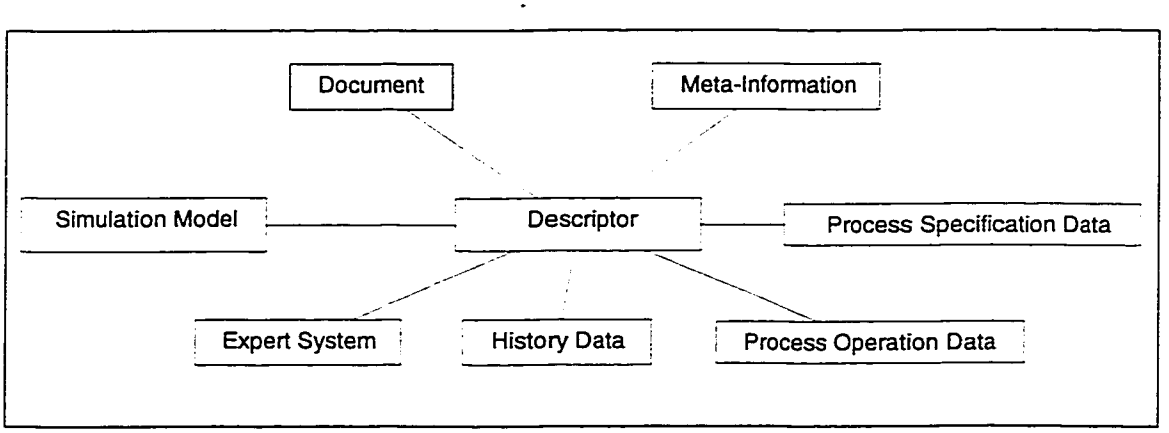


Figure 4 - 6 Overall conceptual model with process specification.

4.2.2 Conceptual model of the process operation data

What are process operation data (POD)? How does one record process operation data? A conceptual model of what the data model should be is of great

importance if these data are collected, stored and manipulated by the IT-DS System. This is the purpose of this section.

The operator observes and/or measures water quality parameters at various locations in a plant to collect information that will be used in process control[2]. This implies that there are at least two entities: *observation* and *measurement* about water, which must be recorded. Observation would include entities such as appearance and odour, whereas measurement would be anything that can be quantified. Information about the locations (called sampling locations) also needs to be recorded with the actual observations or measurements. Moreover, the recording, testing and analyzing of data are usually based on established *protocols*, or standard methods maintained by a governing body such as the Water Environment Federation (WEF). POD also contains the times at which they are taken. There are usually two time points associated with any observation or measurement: a time at which the physical event happens and the time at which the observation or measurement is taken, hence there is a dual time nature of the record that must be taken into account. Furthermore, in the diagnostic process, an observation or conclusion may be incorrect, so the POD must also document the wrong observation along with the correct one. These are some of the issues that the model for POD must be able to handle. Moreover, the model must be flexible to handle unforeseen changes in future requirements.

---

[2] Some of these data are automatically recorded and processed by a Supervisory Control And Data Acquisition (SCADA).

A suitable pattern to deal with these requirements is the *observations and measurements* analysis pattern (OMAP), developed by Fowler (1997). This pattern is derived from the modelling of a healthcare system that deals with the problem of recording and providing access to patient's clinical information. The issues addressed by Fowler's pattern proved to be a superset of the issues identified so far; therefore, this pattern can be adapted and reused very well.

The OMAP is a conceptual object model and involves concepts such as *quantity, unit, conversion ratios, compound units, measurement, observation, protocol, dual time record, rejected observation, active observation, hypothesis* and *projection.* The author also proposes the concept of *associated observation,* used in documenting a diagnostic process to record the evidence and the knowledge that was used for the diagnosis. The object that is the subject of the observations and measurements is called *object of care.* In Fowler's case, it is a person. In this work, it is a water sample.

The *quantity* object was needed to express that a measurement has an amount and a unit. An amount can be an integer or a real value. A *unit* is an object and can be atomic or compound (derived type). However, a measurement using quantity alone is not very useful. How does one differentiate between the measurement that represents COD from that of BOD? A concept known as *phenomenon type* proposed in OMAP will be reused here to provide the differentiation. Using this concept, a water sample has a measurement of x quantity of phenomenon type COD (*i.e.,*

measurement is COD (phenomenon type) of value 100 mg/L (quantity)). This model is shown in **Figure 4 - 7**.

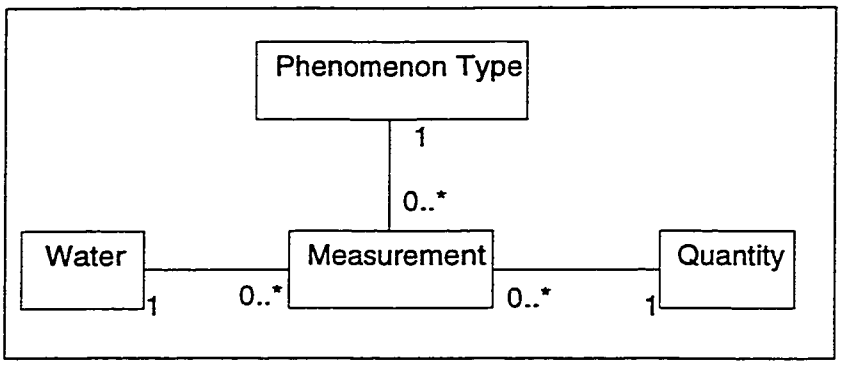


Figure 4 - 7 Water measurement and phenomenon type.

Operators frequently log qualitative observations about water such as color and odor along with measurements such as COD and BOD. One can say that a measurement is a quantitative observation and is a specialized kind of observation. An equivalent qualifier object to *quantity* must be made available to classify qualitative observations. The OMAP suggested *phenomenon* object to qualify a *category observation* (qualitative observation). The phenomenon type is also used to distinguish among category observations. For example, the operator can record two category observations about a water sample: the first one has as phenomenon type **color** with phenomenon **yellow**, and the second one has as phenomenon type **odour** with phenomenon **stingy**. The use of the explicit term phenomenon helps to separate quality from quantity. A model showing both measurement and category observation to derive from a general observation class is shown in **Figure 4 - 8**.

Every observation made must be associated with a time point; however, an observation can be valid only during a certain time period. One such use in the wastewater treatment plant setting is the recording of a start time when a problem has occurred and the time duration of the problem. This dual time record is also useful in the case when a sample must be sent for analysis. To accommodate this, OMAP proposed adding a *time record* object to the observation. This time record object has a time point object and a time period object. These two components encapsulate the recording time and the temporal applicability of the observation. It can also be used to record the time that a water sample is collected and sent for analysis and the estimated time period before a result is entered.

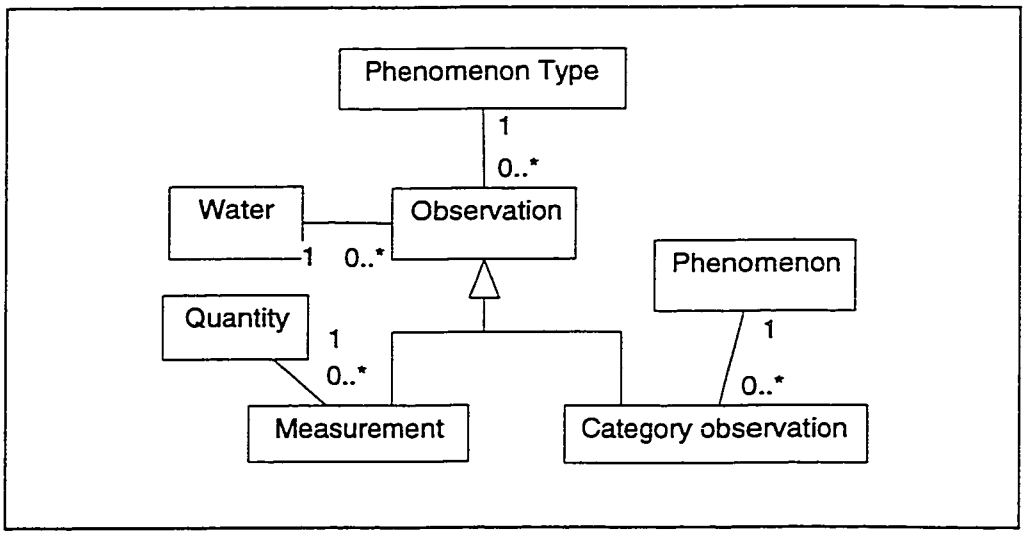


Figure 4 - 8 Water with measurement and category observation.

Several derived versions of an observation can exist. In OMAP, an observation is always abstract (**Figure 4 - 9**) and is represented in concrete form as a hypothesis, a projection, an active observation (current belief), an associated

observation (observation accompanied by evidences) or a rejected observation (incorrect diagnostics). Each observation encapsulates the data and behaviours to provide the appropriate semantics. It also specifies a specific interface so that other objects can distinguish and use each observation correctly. These kinds of observation are not very useful in the tracking of water quality, where simple qualitative and quantitative observations were sufficient. However, they are useful in recording process history data as in the case-logs and hence will be used in the history data model.

One last contribution of the OMAP is the concept of *protocol* associated with each observation. A protocol in OMAP is a record of the process used to make the observations. In recording water quality data, this concept of protocol can be extended to allow the verification that a protocol has been followed for a given observation. The protocol concept of OMAP could also be extended to provide analysis of data; it could also be providing documentation explaining the steps or rationales for each step to the operator *via* an association with the document management package.

The appropriate protocol can be created using a *protocol factory* object. A protocol factory is based on the *abstract factory* pattern documented by Gamma *et al.* (1995). For example, a protocol factory based on name creates an analysis protocol. Once created, it will ask for appropriate measurement observations as inputs and then return the results in a suitable holder object for processing by a presentation

system. Alternatively, a measurement protocol is created as a new measurement is being entered; this measurement protocol will enforce whatever constraints it contains on the measurement being entered.

The comparison between the actual data and some projected data (*e.g.*, permit limit or simulated results) is frequently required. Several parameters such as the time period, the unit-process or unit-operation of interest, type of data and others are required. These parameters are put together by an object, which encapsulates the range of data under consideration and the type of calculation required. This object can be a client to the analysis protocol described above, or it could start a simulation process, get the results and then invoke a comparative calculation protocol. This objects is called *status* object in OMAP, and it is responsible for having the necessary data and for telling what type of calculation is needed. The term *status* was used by Fowler to denote that this object checks on the status of the data as well as the calculation protocol. However, this terminology is rather cryptic. For this work it will be called a *comparator* and the responsibility of a comparator is the same as that of the status object.

The *status* object in OMAP uses a helper object called *range*. A *range* is proposed as an object because one may need to know such information as whether a particular value is within a range, whether two ranges overlap, whether two ranges are adjacent to each other, or whether a set of ranges form a continuous range. **Figure 4 - 9** summarises the conceptual model that will be used in recording process

data for a wastewater treatment plant, where the respective class notation represents all comparator and range objects. Note that the classes observation and protocol are annotated as abstract classes because they can not exist alone; the derived classes are those that have the real presentations. The abstract notation is necessary to show the relationship of measurement and category observation, or how a protocol would be created and used.

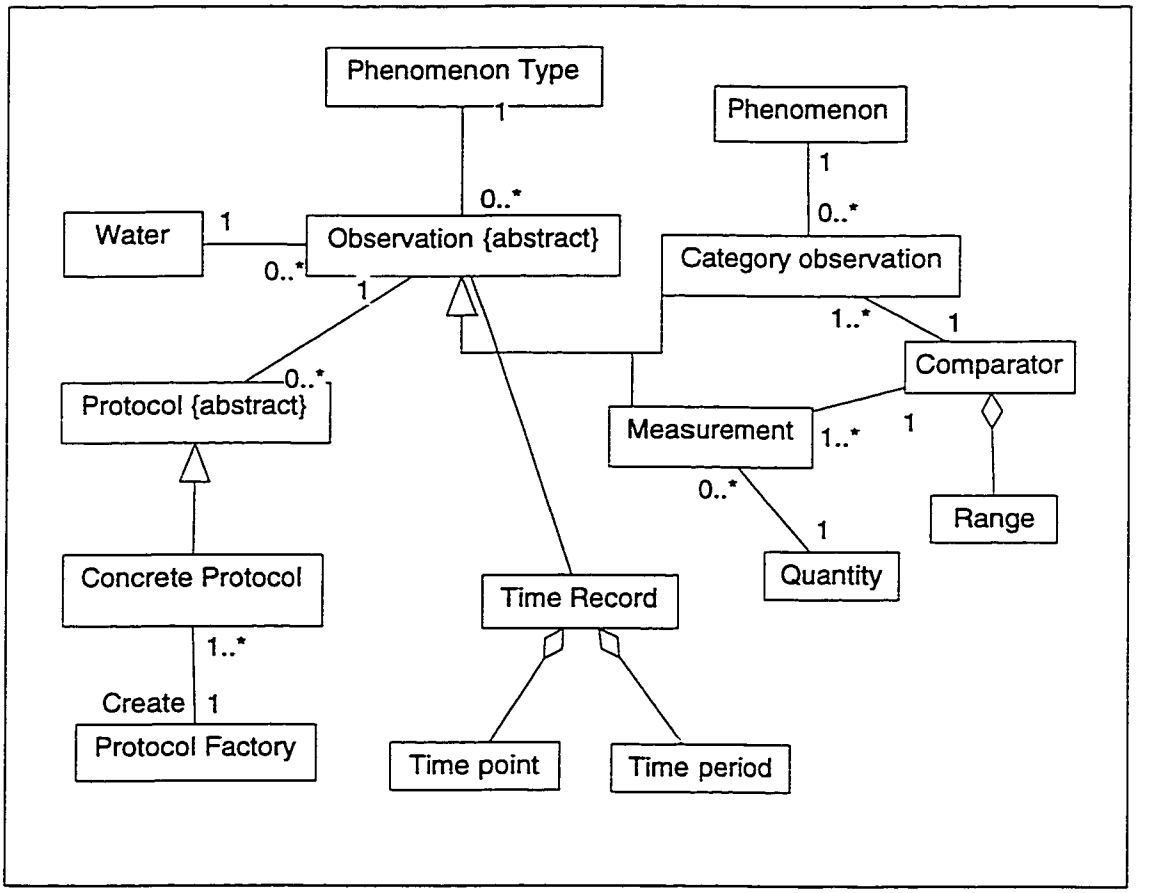


Figure 4 - 9 Conceptual model of water process data.

4.2.3 Document conceptual model

There are two types of document management. One is concerned with the

support for cooperative works of a team of authors, publishers, and editors. The second type of document management focuses on the whole document and is concerned with the collection, organization and distribution of documents. In the first type, parts of a complete document may be shared, concurrently processed, versioned and put together for production. An example of the second type is an electronic mail system. The second type also covers the field of electronic document management system (EDMS), in which the system helps the user in archiving, searching, and locating electronic documents (Koulopoulos and Frappaolo, 1995).

Document management is a large topic with many ramifications and issues that would take several chapters to cover. In this work, a simple model of a document to support the on-line process documentation is proposed. The conceptual perspective of this model is concerned with the collection, organization and retrieval of an abstract document and any of its parts. Some first classes of objects of interest are *DocObject*, *DocPart* and *AuthorParty*. This first model is shown in **Figure 4 - 10**.

This model is a variation of the *composite* pattern described by Gamma *et al.* (1995). All, except *DocObject*, are abstract classes. This allows specialization of responsibility *via* sub-classing. An *AuthorParty* is responsible for the identification of document(s) authorship. The concept of a *party* is discussed in the work of Fowler (1997). Here, the term *party* is used to generalize information about author(s) and publishers since the identities rather than the roles of these entities are of interest. A

DocPart can be a section of text, a picture, or a piece of video or audio. It can even be an interactive, embedded executable component (*e.g.*, a Java Applet or an Active X control). A *DocPart* can contain other *DocParts* (a *DocObject* in this case); it also knows how to render itself or create an intermediate form that can be rendered. A *DocObject* knows all about its *DocParts* and how collect them and lay them out. To perform layout, a *DocObject* needs to know about the logical organization of its *DocParts*.

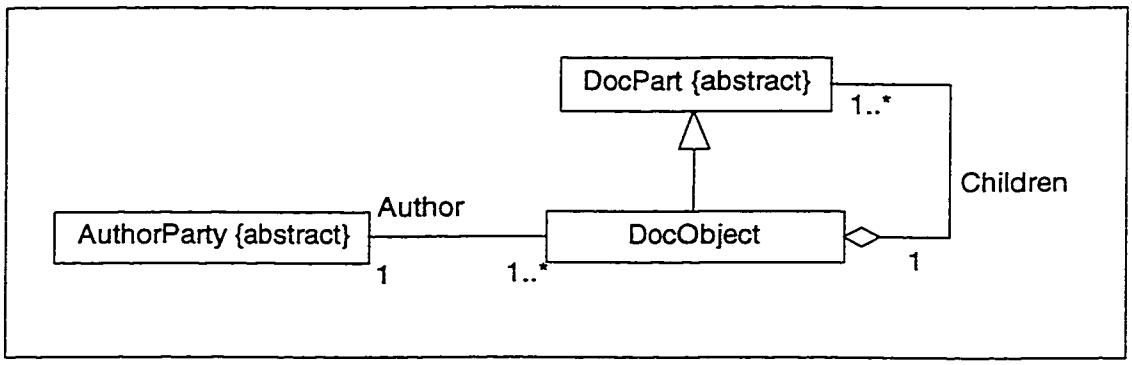


Figure 4 - 10 First document management conceptual model.

The logical organization has two levels: a semantic level and a physical level. The semantic level contains structural information that is meaningful to the human audience that can be spatial, temporal, hierarchical or in other formats (Burnett, 1990). In addition, navigational structure (document meta-information) that can be used in orientation is presented as a book's index, table of content, frequently used items and others. Physical layout of the document parts (*e.g.*, position on the page) and document location marker (*e.g.*, a page number in an index) are frequently used to give a spatial sense to the reader. In the computer, the physical layout can be

expressed as screen position, order of display, time of display or event-trigger of display of document parts. The computer also supports meta-information such as table of content and index by providing a graphical representation of the document's various parts that can be interacted with to bring the audience instantly to the part of interest.

In light of these complex tasks that must be handled in a layout activity, it makes sense to create a new abstraction called *DocLayout* as an abstract class. *DocPresentation* is a concrete class derived from the abstract class *DocLayout.* It contains information that helps a *DocObject* to layout its *DocParts* on the screen or in an alternative format, and it informs the *DocObject* when to begin the presentation and other instructions. Furthermore, *DocObject* is associated with its specialized version of meta-information: the class *DocMetaInfo,* which it connects indirectly *via* the *descriptor* object (see **Section 4.2.1**). This class keeps the meta-information about the document to allow the construction of the table of contents and index and to support searching and querying functions. The resulting augmented document model is shown in **Figure 4 - 11.**

4.2.4 Meta-information conceptual model

The meta-information (meta-data) keeps the summary information about other objects that the user can activate to obtain further functionality (*i.e.* in a one to one relationship). This class of objects also needs support for search and query and creation and rendering. **Figure 4 - 12** shows the conceptual model for the meta-

information, where these meta-data are associated with four other classes: *AuthorParty, Keyword, Context* and *Abstract.*

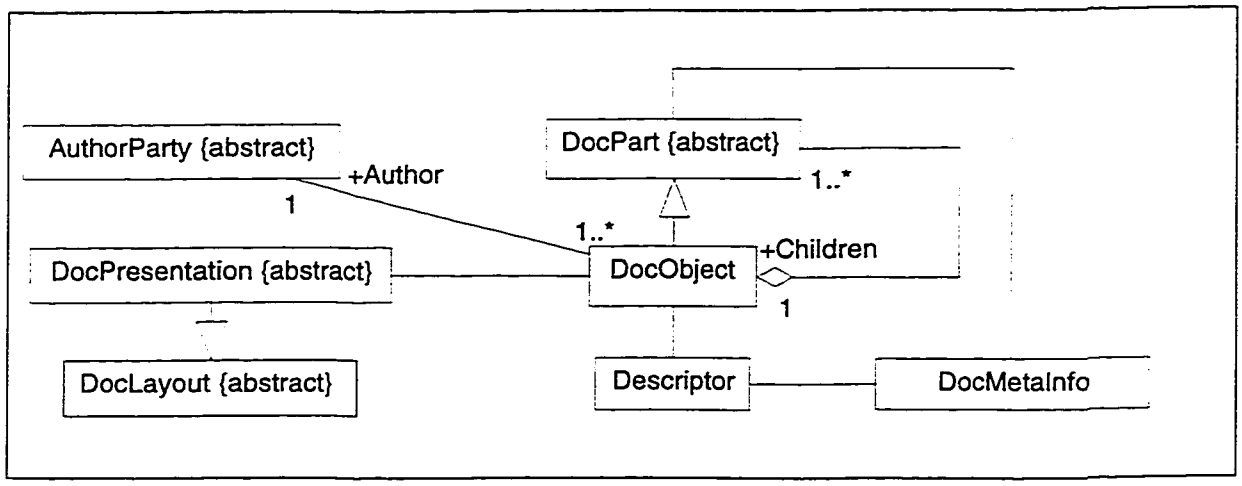


Figure 4 - 11 Document conceptual model with layout abstraction.

The meta-data for each type of object are formed by including an instance of the class MetaData as an internal structure (**Figure 4 - 12**); this is referred to as modelling by composition (Coad and Mayfield, 1997). These are represented by *ESMetaInfo* for Expert System, *SimMetaInfo* for simulation model, *CLMetaInfo* for history data (*i.e.* case-log) and *DocMetaInfo* for document. The *context* object mediates the locating and activating of the object of interest. This action includes tasks such as initializing the run-time environment for simulation, expert system, and getting history data and document. Note that context is an abstract class; therefore, *SimContext, ESContext* and *HistoryContext* are the actual classes that know how to deal with each type of object.

4.2.5 Conceptual model of expert system and simulation model

The model shown in **Figure 4 - 12** did not specify how the different context objects would interact with history data, simulation model or expert system. In the case of history data, the interaction is straightforward because the history data format (model described in the next section) does not change. However, the situation is more complicated with the simulation model and expert system. There are several simulation models with different sets of required parameters and they can potentially produce different outputs.

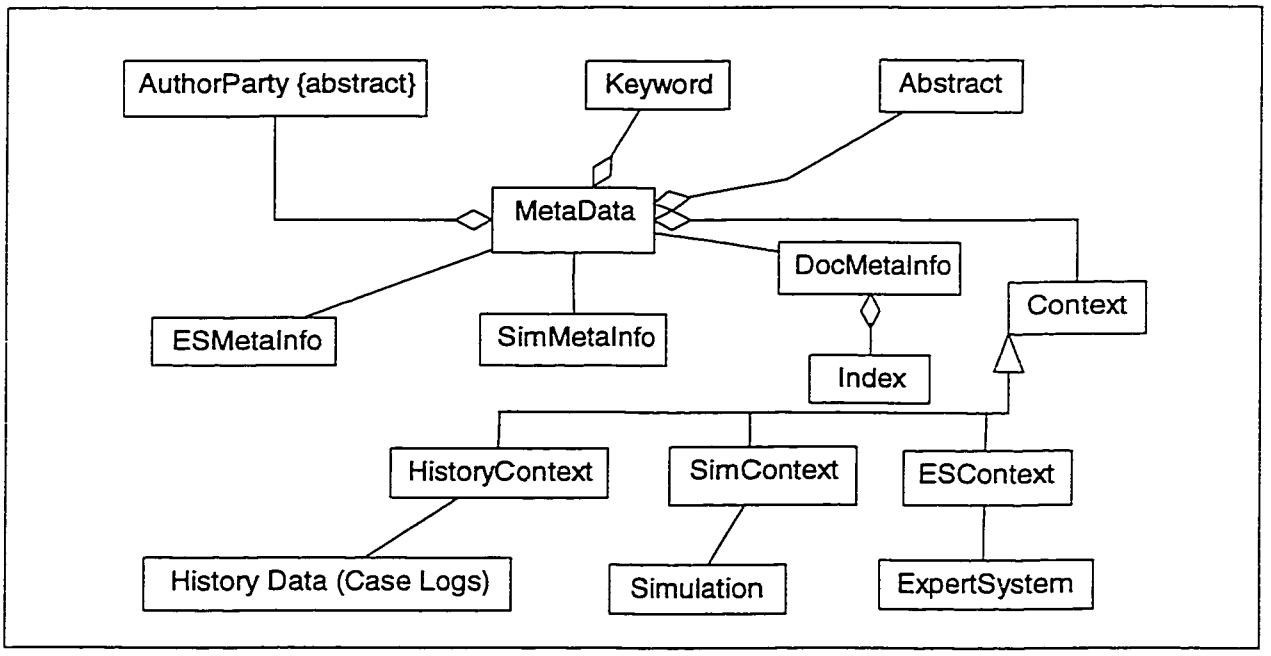


Figure 4 - 12 Meta-information conceptual model.

Providing a context for each model is one solution. However, as the number of simulation models adds up, the number of contexts will become unmanageable. Furthermore, in the case of expert system there are currently two systems of implementation: a stand-alone program or using a combination of an expert system

shell and a shell script. In the case of a stand-alone expert system, a situation similar to supporting multiple simulation models emerges. With an expert system shell, the context object not only has to initialize the run-time support for the shell, but it also has to know how to retrieve the correct shell script. In all cases, a better way than implementing customized context object must be found. In order to be useful, it is required that the creator of simulation models and expert systems must conform to agreed interfaces, limit the system to support only a number of well known expert system engines or simulation tools.

To deal with this situation, the *strategy* design pattern (Gamma *et al.*, 1995) is used. This pattern was found in solving the problem of related classes differing only in their behaviours. It suggests defining an interface common to all support algorithms in a class called *Strategy*. A *Concrete* class uses this interface to call the algorithms defined by a *ConcreteStrategy* class, which is a concrete sub-class of *Strategy*. This pattern is illustrated in **Figure 4 - 13.**

In this work, the *Strategy* pattern is used twice because of different interfaces between the simulation model and expert system. To further accommodate the need to add specialized interfaces, the interface will not be a method (or group of methods) as in Gamma *et al.* (1995). Rather, it will be a class by itself so that specialized methods can be extended *via* inheritance (*i.e.* an interface class). Furthermore, the interface class will be composed of three other classes to separate the essential responsibilities: *InputInterface* to manage input parameters,

OutputInterface to return results and *ConfigurationInterface* to configure fixed parameters. However, this model is inadequate to handle special requirements such as expert system shell script. Moreover, the use of the three interface classes is rather limiting.

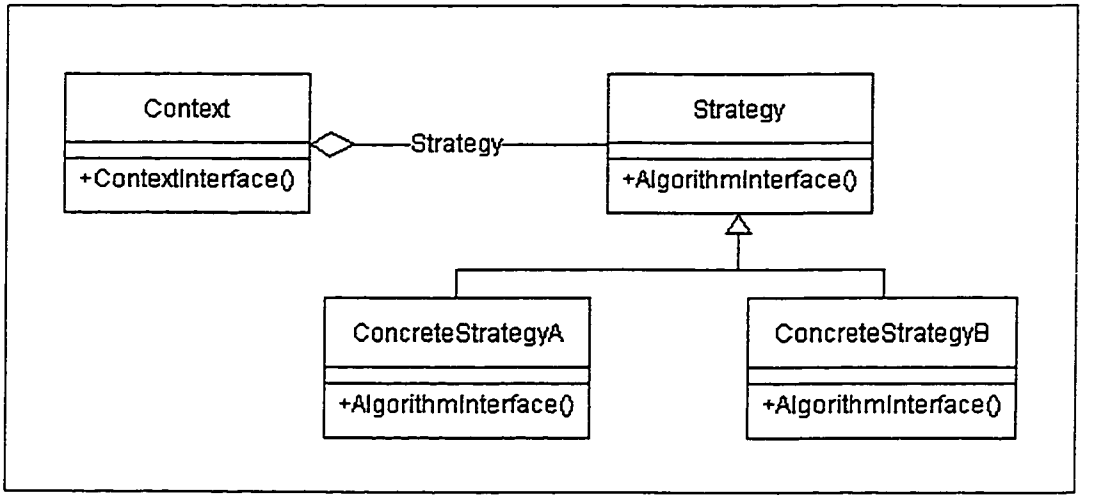


Figure 4 - 13 The *Strategy* pattern (Gamma *et al.*, 1995).

To deal with this limiting situation, three new classes: *InputParameter*, *OutputResult* and *ConfigData* will be added to the model. In this new model, the expert system designer and the simulation model developer need to develop concrete classes such as *InputInterface*, *OutputInterface* and *ConfigurationInterface* to provide implementation specific information on how to interpret respectively the *InputParameter*, *OutputResult* and *ConfigData* that they supply. The IT-DS System will interact with the expert system and simulation based on the standard interface classes, hence providing certain flexibility in allowing multiple simulation models

and expert system to be utilized. The resulting model for dealing with the expert system and simulation is shown in **Figure 4 - 14.**

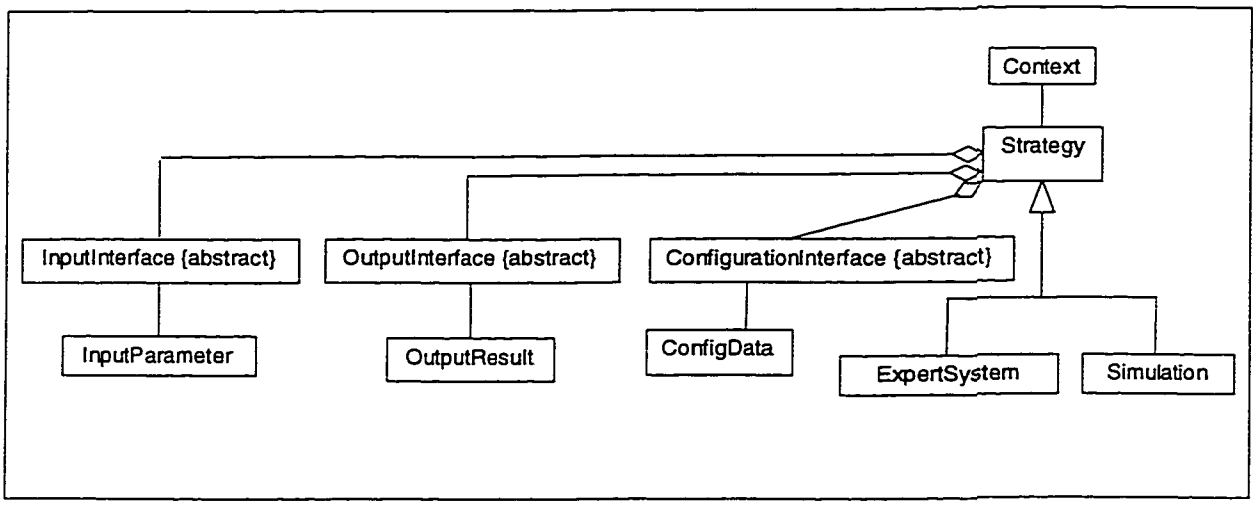


Figure 4 - 14 Conceptual model to handle simulation model and expert system.

4.2.6 Conceptual model of history data (case-log)

Historical data about a process are used mainly to document a failure and associate recovery or any observations that would help the operator learn more about the process and take advantage of past experience. This is the rational behind keeping logbooks at the plant. Unfortunately as presented in Chapter 1, this written form of logbook is not used often due to potential tedious manual search or browsing for information. For an experienced operator, this does not present a problem; however, the plant may experience hindrance if this experienced operator leaves. Providing ease of access to this valuable source of information is the goal and responsibility of the history data in the IT-DS System.

A history data model can be complex because it would have to accommodate water records, information about the unit-process, charts and even document. History can be understood to be everything that has ever been recorded about a unit-process. In this work, history data consist of a collection of case-log objects. Each case-log represents a closure about a failure or an event of significance that has happened to the unit-process.

In **Section 4.2.2**, a conceptual model of how information about the treatment of water (*i.e.*, water sample) could be kept has been developed. At the same time, concepts such as hypothesis, projection, active observation (current belief), associated observation (observation accompanied by evidences) and rejected observation (incorrect diagnostics) were formulated as possible concrete classes of the abstract observation class. In many respects, the case-logs that made up the history data can be considered as a more specialized version of process operation data (POD).

The POD model in **Section 4.2.2** could be expanded to include all the extra concrete classes of the abstract observation class. However, a model for a case-log that is simply based on this expansion is incomplete because the process identity containing several water samples that is documented in the case-log. A case-log also needs to record time, and the time record class developed previously in **Section 4.2.2** can be reused here to denote both the time of occurrence and the duration (*i.e.*, instances of the class time point and time period respectively). A better model

would include a connection to the relevant unit-process (the subject of the case-log) and a collection of water data. The new model is shown in **Figure 4 - 15.**

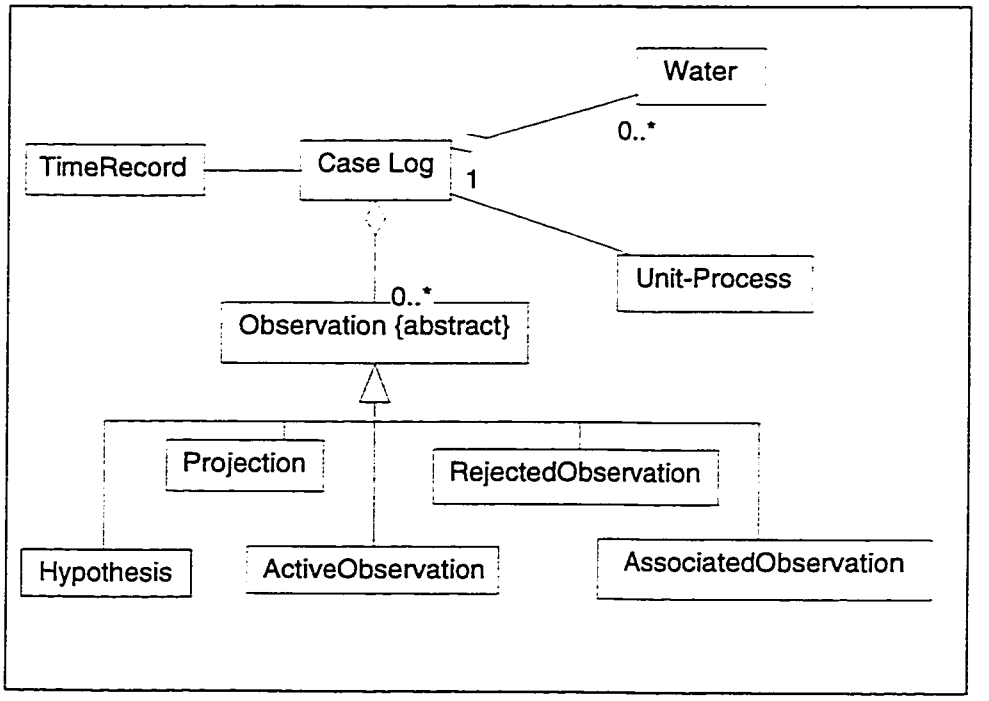


Figure 4 - 15 Conceptual model of a case-log (history data).

4.2.7 Conceptual model of wastewater treatment plant

In previous sections (**4.2.1, 4.2.5** and **4.2.6**), a need for abstraction that would capture information about the limits of a unit-process and others has been stated. This kind of information is often needed to initialize simulation models and expert systems. It also serves as a more complete identification for the unit-process that is referred to in case-log, POD and document. Furthermore, these abstractions are needed to set constraints on how the unit-process, the unit-operation, the receiving water and others are to be arranged and composed.

Another need for such abstractions concerns with the representation of different functionalities of the system to the operator. It is more natural for the operator to interact with familiar objects. One such scenario involves the operator accessing a unit-process document *via* its graphical representation. The relationships and entities in the physical domain of the unit-process, unit-operation, receiving water, collection source and others are needed to establish rules governing how the corresponding graphical representation would be put together.

The wastewater treatment plant receives wastewater from a collection system, processes the wastewater through its unit-processes that it has and discharges the treated water into a receiving water body such as a lake or a river. The employees of the plant manage and operate the plant; they are the plant's manager and operators. The operators are the users of the IT-DSS. There is also a special employee who will be the administrator of the IT-DSS. Based on this initial description, a wastewater treatment plant object, a collection system object, employee objects, unit-process objects and a receiving water body object can be identified.

Within each unit-process, there are one or more unit-operations. The unit operation is a piece of equipment that is necessary for a given unit-process to do its work. For the activated sludge unit-process, the basic unit operations are the aeration tank and the secondary clarifier. Each of the unit-operation can then be further divided into several individual parts. The association between a unit-

process and its unit-operation(s) is a strong aggregation relationship. This kind of association also exists between the unit-operation and its parts. From this analysis, a unit-process object is composed of one or more unit-operation objects, and each unit-operation object is composed of one or more parts.

Water is tracked from the moment it enters the plant until it is discharged into a receiving water body. It has different names as it progresses through the plant's unit-processes, but only its characteristics are of vital importance to the plant's operators. The process data management package records and manages information about water. Water is treated as a class of object from which the instances represent the various states of water as it passes through the plant. The resulting model for the domain of a wastewater treatment plant is presented in **Figure 4 - 16**.

Only the class names are shown in **Figure 4 - 16** because only the associations are of interest. Also, not shown due to space limitation are the inheritance associations between unit-process class and its specialized classes, between employee class and its specialized classes, and between unit-operation class and its specialized classes. For example, the sedimentation and activated sludge processes are two specialized classes of unit-process, operator and manager are specialized classes of employee, and aeration tank and clarifier are specialized classes of unit-operation. There can also be specialized classes derived from the general part class, but this relation is not important within the scope of this work.

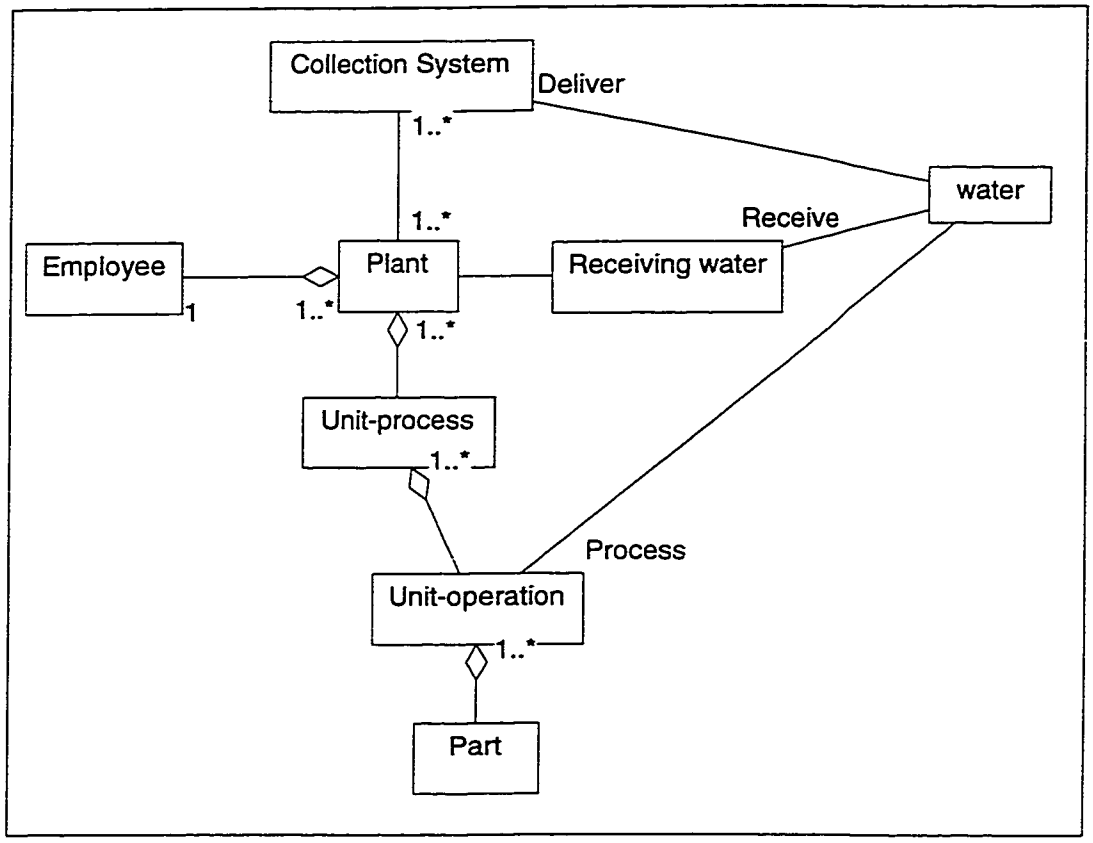


Figure 4 - 16 Domain conceptual object model of wastewater treatment plant.

4.3 Specification models

Several possible systems can be built from the conceptual models that have been presented; however, the complete specification of all conceptual models and consequent implementation of one complete system is not possible within the time scope and resources available for this work. For this reason, this section will only present a set of specification models for water records, meta-data, history data and document, while leaving the rest to future works. These specification models describe a set of possible operations of each class, which are usually referred to as

the interfaces of the system. How these interfaces will be implemented is the subject of **Chapter 5.**

4.3.1 Specification model for common services

As the use-case demonstrated clearly in **Figures 4 - 1** and **4 - 2**, the system administrator and operator spend a great deal of time creating new objects, storing them permanently into the system and conducting direct or indirect searches for these stored objects. These activities are performed to make the objects do some useful work or to modify their contents. To support these activities, the system would have four categories of services available: support for object creation, support for object persistence, support for object search and support for object display.

These supports are best modeled as components that work between the user interface and the back-end services provided by the database management system, third party search engine and so forth. These component models and their interactions are shown in **Figure 4 - 17**. Note that the search, data manager and class factory are part of the application server, whereas the GUI system is part of the client package (see **Section 4.1.1**)

4.3.2 Specification model for persistent component of IT-DS

In the following sections the specification models of water record, meta-data, document, case-log and plant specification data will be presented and discussed. As stated previously, the completion of a fully functional prototype of the IT-DS system

is beyond the scope of this work; therefore, only a prototype of the data management portion will be completed. The specification model of expert system and simulation and their prototype implementation are beyond the scope of this project. The following common use-cases apply in data management:

(a) User files a created object (*e.g.*, document, page, expert system module),

(b) User searches for objects,

(c) User modifies objects, and

(d) User updates objects,

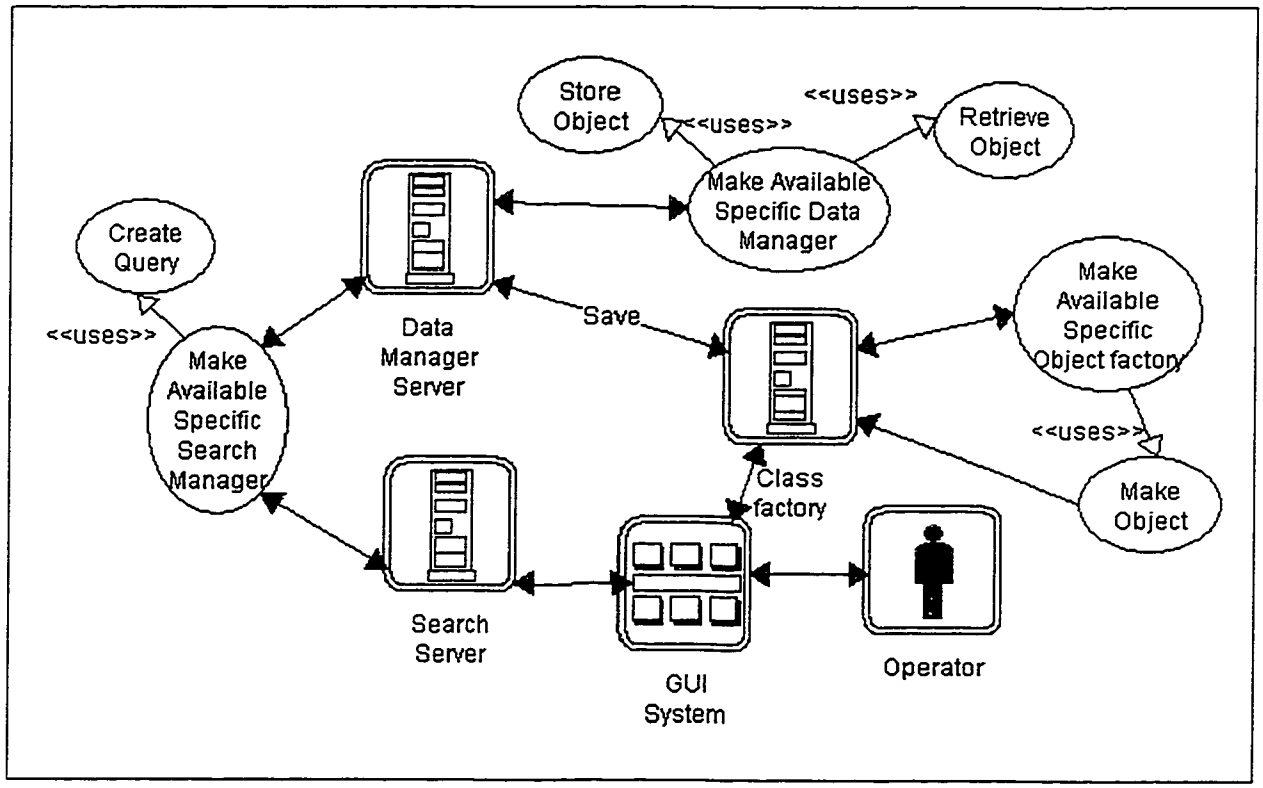


Figure 4 - 17 Common services.

Here, an object can be one of the many objects of the classes discussed in this work. In use-case (a), three scenarios can occur. In the first scenario, a new object is

created, and then it is filed. The second scenario involves filing of a modified object, and in the third scenario the filing operation is cancelled. The filing action makes the object permanent by using a persistent mechanism.

The user wishes to search for a particular object (or objects) in use-case (b). Some possible scenarios are:

i. The user either knows what to look for and can formulate the search parameters (finite search),
ii. The user has an idea that guides the search (investigative search),
iii. The user does not know how to search (serendipitous search, for more details refer to classifications of searching (Koucopoulos and Frappaolo, 1995)), or
iv. The user can cancel the search.

A system feature that would support scenario (i) is interactive query. The basic steps involve creating queries and having the queries somehow processed by the system to return the results. To support scenario (ii), the system could provide a template to collect incomplete knowledge and attempt to use mechanisms developed for scenario (i) to return a range of possibly matching results. In dealing with scenario (iii), the system can actively query the user and then attempt to generate results that can be used to further narrow down the search parameters so that mechanisms in scenario (i) and (ii) can be used. To deal with the last scenario, the system would simply provide options so that the user can cancel the search operation. It can be seen that a basic finite search facility can be used to build more sophisticated techniques, since all forms of search will eventually be a finite one.

A finite search involves the sequence: getting the correct search parameters, prepare a query, submit a query to the appropriate query processor and accept the returned results. A query can be simple or complex and is encapsulated in a query object. The process of creating a different query object can be encapsulated in the responsibilities of a query generator. Once created, a query object must be handled as efficient as possible. This task is the responsibility of a query processor, which knows how to either process the query or delegate the task to the appropriate object. From the operator point of view, the queries can be formulated either *via* a menu item of the GUI or by manipulation of the graphical representation of the unit-operation or unit-process. In either case, the system will route the request to the query generator.

In the second search scenario, support for the guiding idea is possible by allowing the user to browse through a collection of objects. The browsing activity can be full-access to every object, or a large number of objects can be browsed after specifying a general query first. Both approaches can be applied with the support of a general query and so general query will be used in this project.

The situation is more difficult for the third search scenario. A possible solution is to guide the user by suggesting a search strategy based on system-user interactions. A typical example would be that the user enters a phrase about something that may be relevant, and then the system will return with a list of possible keywords or query formulations that can be tried. For this purpose, a query generator object would assume the task of taking an arbitrary input text

fragment and produce a number of query objects that can be tried. The functionalities of this query generator can be added over time.

As for handling mid-process cancellations by the user (scenario iv), the system would check at the end of each step for cancellation requests and respond accordingly comply. This is normally handled by the GUI client package.

For use-cases (c and d) the user would use the same mechanisms outlined above to first search for the objects then the mechanisms used in filing the object into persistent storage will be used to update the object state. The only difference from use-case (a) is that an object already exists instead of a new one being created. The specification model to support these basic use-cases is shown in **Figure 4 - 18.**

As shown in **Figure 4 - 18,** the **ASDBComponent** is composed of two objects: **QueryManager** and **DataManager,** and it implements two interfaces: **IManageData** and **IQuery.** The QueryManager and IQuery collaborate to provide the functionality of the Query Generator and Query Processor proposed previously, while the DataManager interface and the IManageData interface fulfill the connection to the database.

Logical queries are processed by the methods of the IQuery interface and are translated into messages to the QueryManager. The DataManager class of objects and the corresponding IManagedata interface handle the creation and storage of new objects and the update of existing objects. DataManager encapsulates private methods to hide the specific database storage technology from the client. To the client, the methods available in the ImanageData interface are sufficient. Overall,

this specification model represents a simple way to handle the persistence and querying needs of the objects in the IT-DS system, suitable for prototyping purposes. However, more work is needed to improve this model.

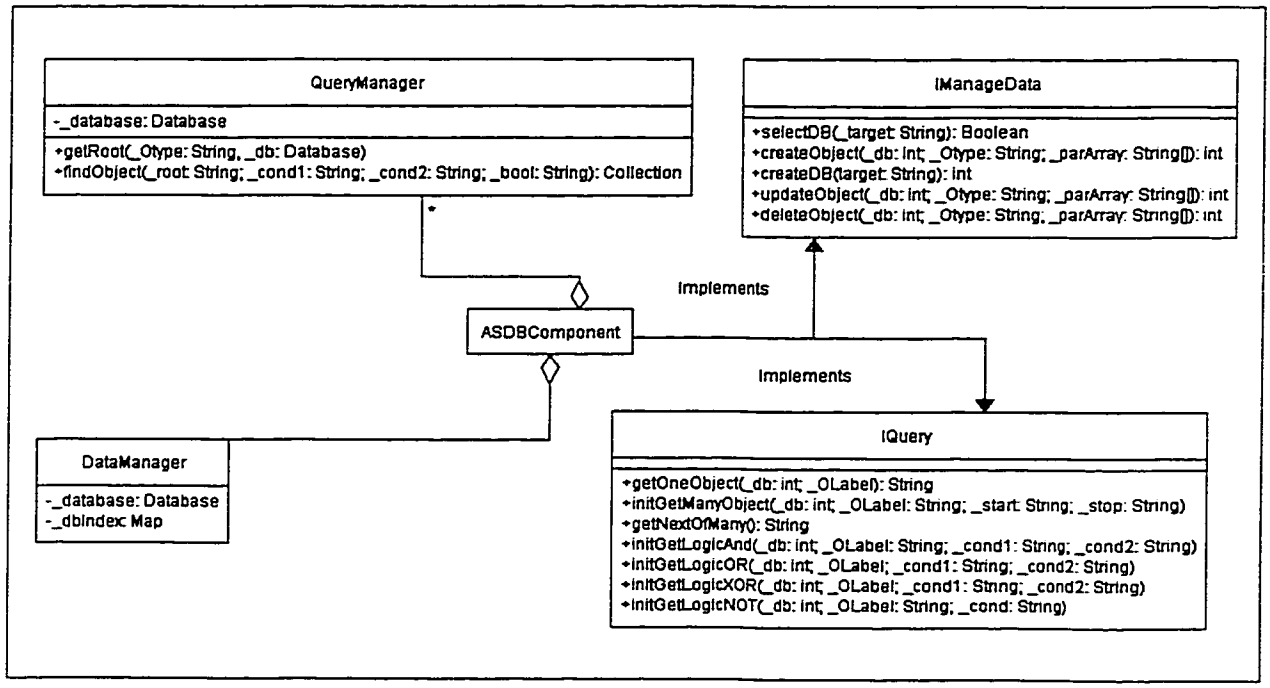


Figure 4 - 18 Database component of the IT-DS system.

4.3.3 Water record specification model

Several decisions have been made to create a specification model for the conceptual model of the process operation data presented in **Section 4.2.2**. The decisions include keeping the attributes of each object private in anticipating future changes, having the *PhenomenonType* class keep the information about associated protocols and documents, and expressing most of the relationships as key strings instead of implicit references. The resulting model is shown in **Figure 4 - 19**, with some of the associations omitted for clarity.

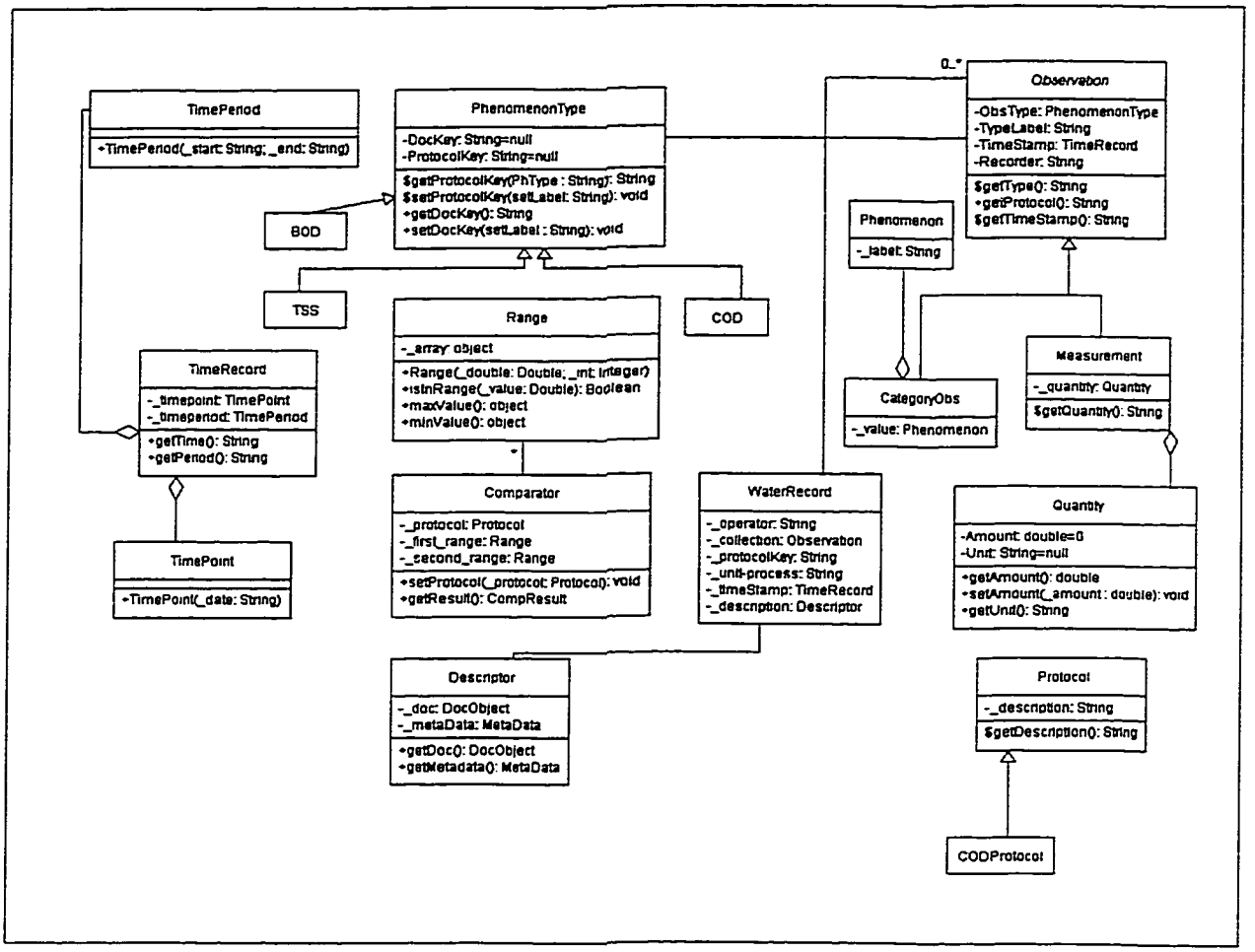


Figure 4 - 19 Specification model for Water Record.

Most of the attributes and methods of the classes presented in **Figure 4 - 19** are self-explanatory. The entities of interest to wastewater treatment are BOD, SS, pH, COD which are presented as examples. Note that these classes are modeled as sub-classes of *PhenomenonType*, based on an assumption that they will be its specialized versions. However, an alternative model exists, in which BOD, SS and other measurements could be modeled as a composition, having a private attribute as an instance of *PhenomenonType*. In the context of the domain of wastewater, the semantic of the measurable quantities such as COD, BOD and SS seem to favor an

inheritance approach, which is the one adopted in this work. However, as more experience is gained in prototype development, a compositional approach may be adopted, where COD, BOD and SS are modeled as roles that *PhenomenonType* could take. This compositional approach is preferred since it reduces tight coupling and provides more flexibility (Coad *et al.*, 1997).

4.3.4 Document Specification model

The conceptual model for document that has been discussed in **Section 4.2.3** is suitable for several kinds of document and their associated format. However, a specification model based on the Hypertext Mark Up Language (HTML) will be presented. Using HTML as a document base has several advantages such as being an open standard, lightweight rendering tools and the pervasiveness of browsers that could be used as document rendering tools. Choosing HTML as a starting point also opens the opportunity for future expansion with emerging standards such as Dynamic HTML and the proposed extended mark-up language (XML) (URL1). Since HTML is a Standard General Mark-up Language (SGML) application, it will also give the advantage of being able to store document in a presentation independent manner, a plus for a document base that is potentially managed frequently such as the one used in a wastewater treatment plant. Aside from choosing HTML as a document base, the remaining features of the document conceptual model are all incorporated with the resulting model shown in **Figure 4 - 20.**

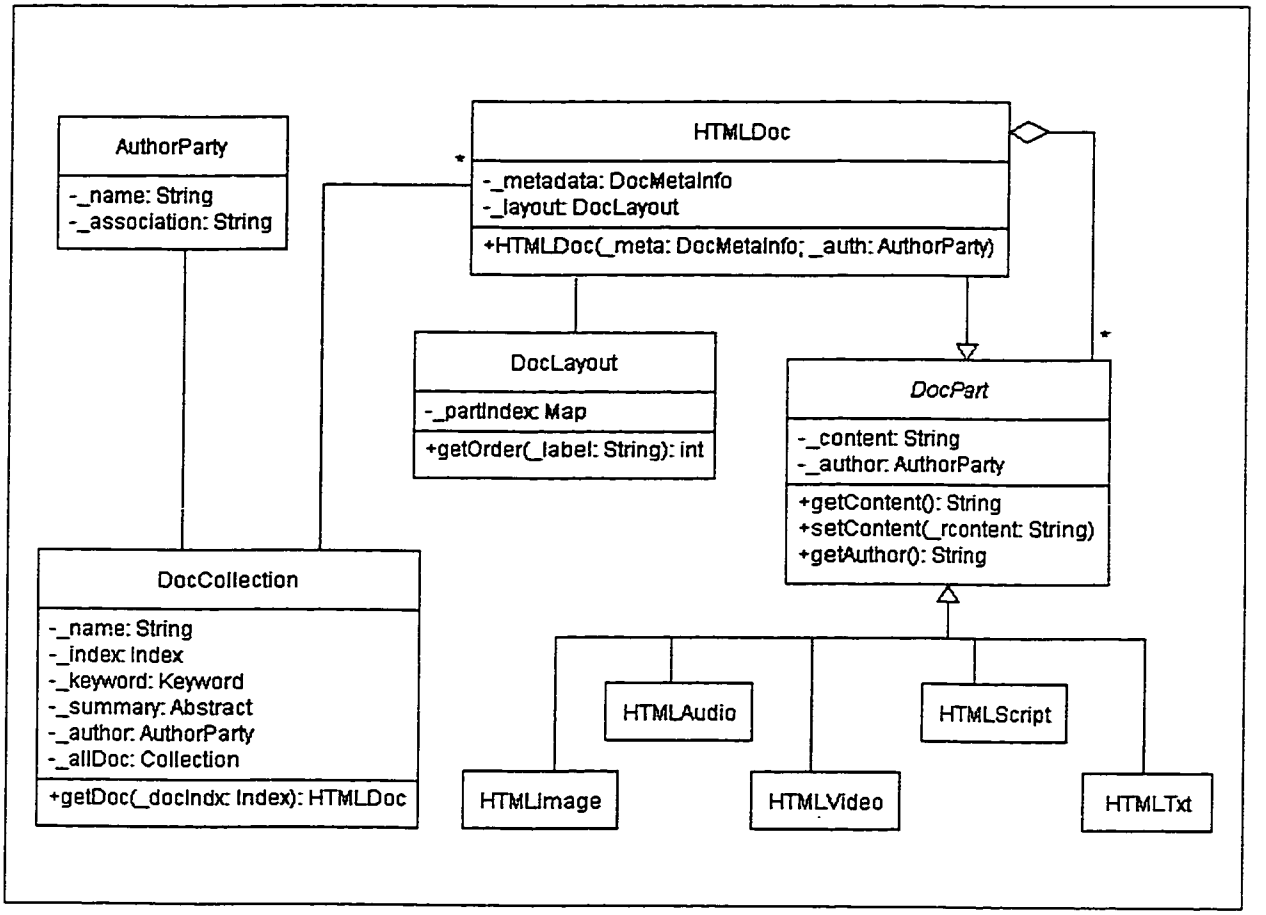


Figure 4 - 20 Specification model for Document.

Document collection was not initially present in the document conceptual model, but it is needed to deal with many documents. There are several classes that are used here such as Index, Abstract, AuthorParty and DocMetaInfo that are part of the meta-data model discussed in the next section.

4.3.5 Specification model for meta-data

The specification model presented in this section is similar to the meta-data conceptual model presented in **Section 4.2.4**, except for one notable difference in

that the specific context classes are absent. The resulting specification model is shown in **Figure 4 - 21.**

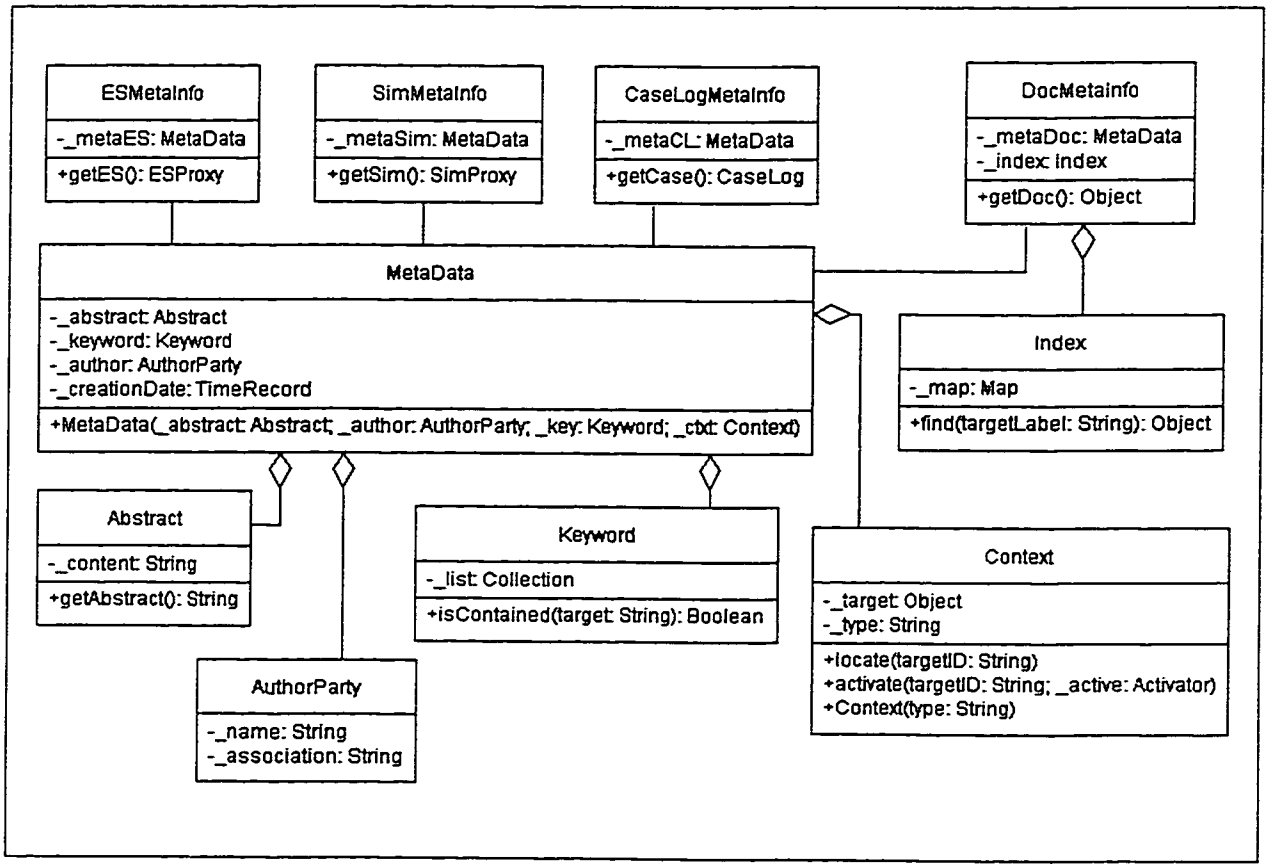


Figure 4 - 21 Specification model of meta-data.

As proposed previously, the classes *DocMetaInfo, ESMetaInfo, CLMetaInfo* and *SimMetaInfo* are modeled as composition of meta-data class, where specific methods to the need of each class will be translated into private methods that were called to the private meta-data instance. Since the context class is encapsulated within the meta-data class, when a client sends a message requesting the actual object of intent (*i.e.* a simulation model or expert system loading), it is delegated to the tasks of the private context instance of the meta-data class variable. The details of how the

activation will be modeled and implemented are beyond the scope of this project. For the purpose of this project only messages sent to *DocMetaInfo* and *CLMetaInfo* will result in an actual return of the document and the case-log.

4.3.6 Specification model for case-log

The case-log specification model is shown in **Figure 4 - 22**. In this model the case-log class is responsible for producing various observations that it keeps. Although not shown, these specialized observation classes are sub-classes of the *Observation* class discussed previously in **Sections 4.2.2** and **4.3.3**. Thus, they inherit the ability to apply classification to observations (*via* the *PhenomenonType* class variable), to keep a record of the time of observation and the identity of the person or entity that recorded the observation. The sub-classes of observation extend the base class further by providing the means for adding descriptive texts instead of quantitative or qualitative observations. The model can still be improved by adding other capacities such as keeping a pictorial, sound or video record about the problem being recorded. However, these additions will undoubtedly complicate the model further. They are deemed not yet necessary in this phase of work and will be of great interest in future research.

4.3.7 Specification model for plant data (process specification data)

The needs for plant data such as reactor size, equipment dimension, equipment capacity and the relationship among the various components of a typical wastewater treatment plant have been discussed in **Section 4.2.7**. The specification

model presented in this section will add another use for the classes that have been modeled previously (**Figure 4 - 16**). This is to mediate user access to various operational parameter used in the activated sludge process. These results include parameters such as the food to microorganism ratio (F/M ratio), the mean cell retention time (MCRT), the hydraulic residence time etc.

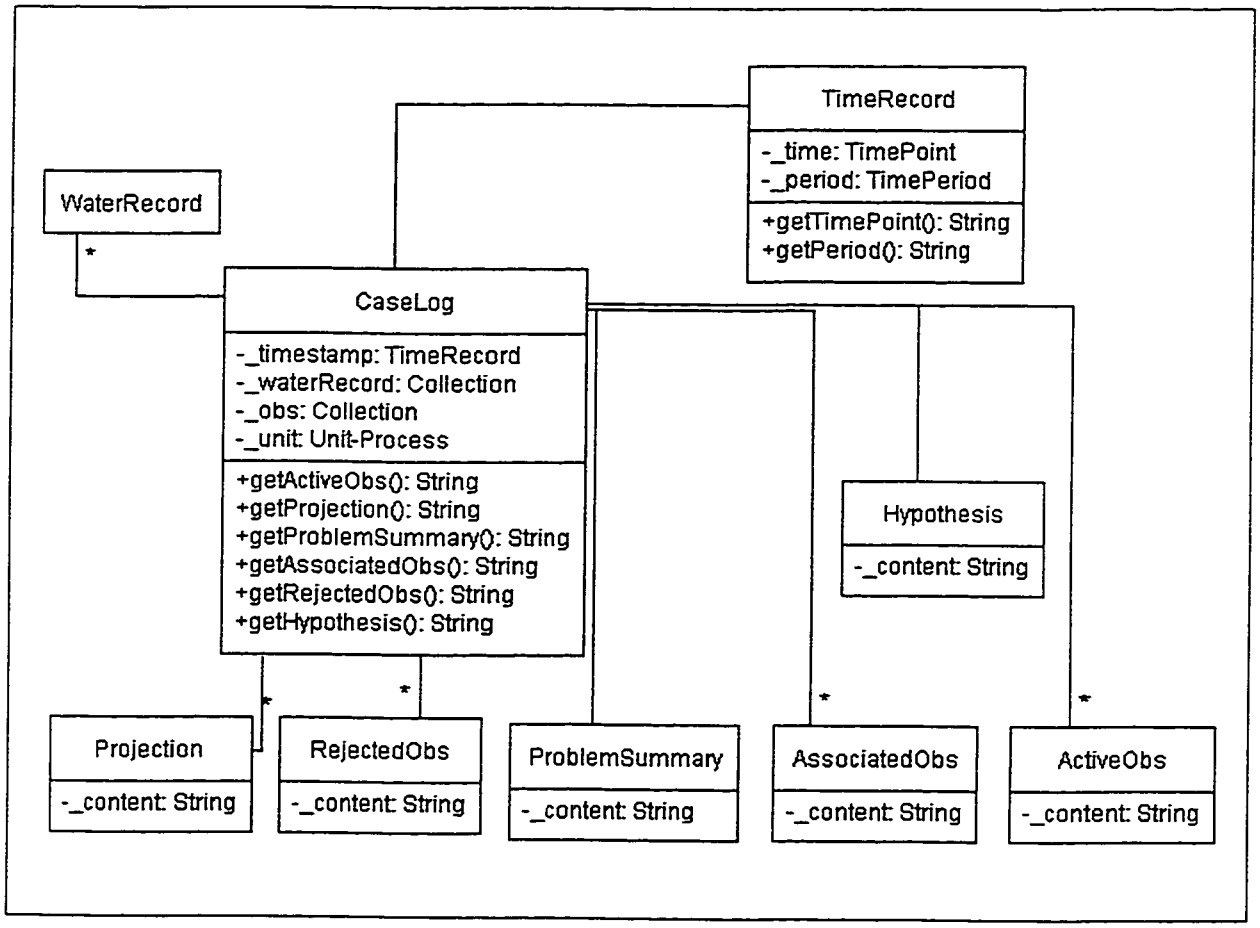


Figure 4 - 22 Specification model for case-log.

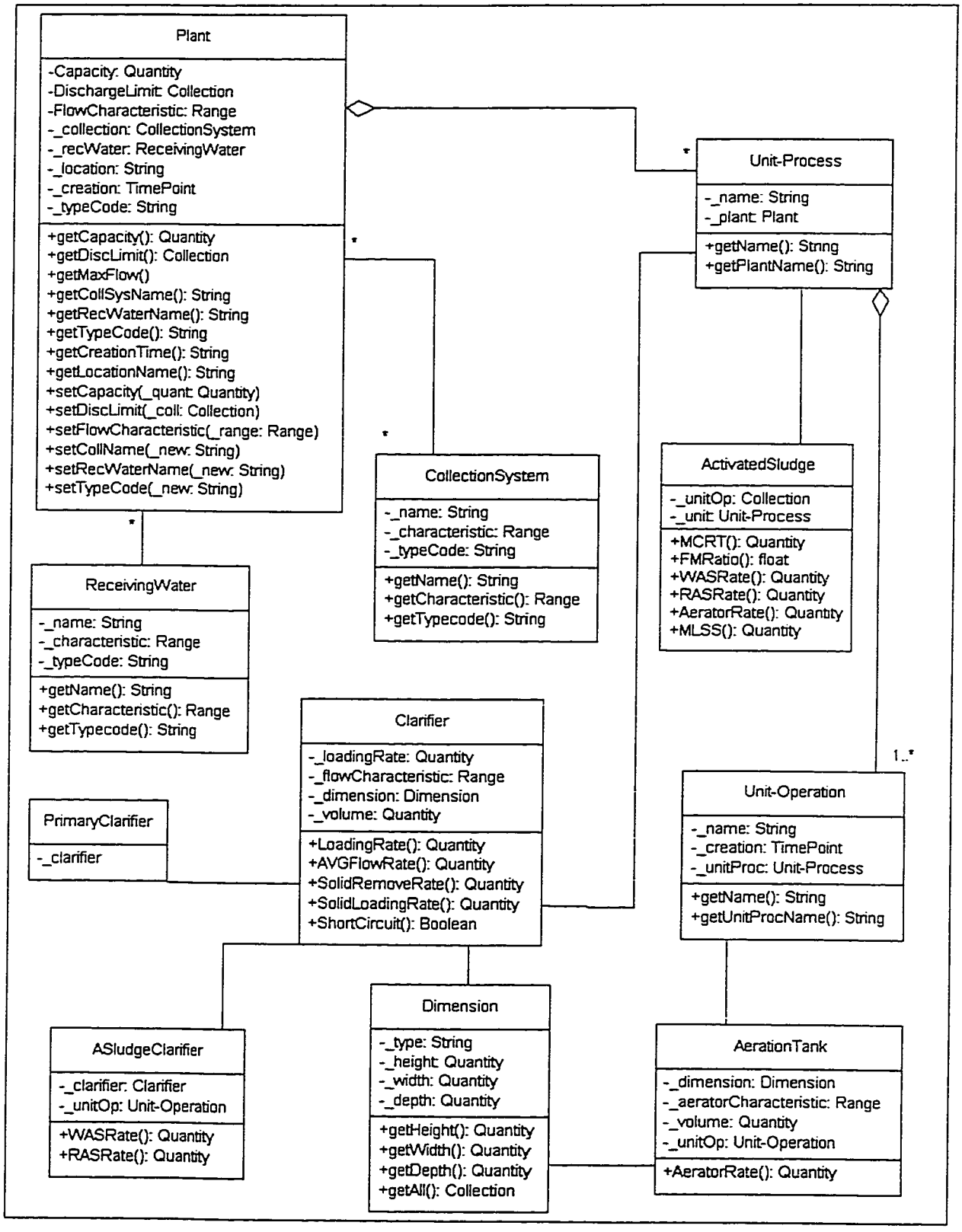


Figure 4 - 23 – Specification model for plant data

To the client these results are available as method calls on the *ActivatedSludge* object as shown in **Figure 4** - 23. As discussed previously, the analysis result can be obtained by asking the service of the proper *protocol* object (see **Section 4.2.2**). However, the client of *ActivatedSludge* object does not have to be aware of how the work is performed. The message to *ActivatedSludge* can be handled directly or by delegating the task to a *protocol* object.

Chapter 5

IMPLEMENTATION OF A PROTOTYPE SYSTEM

A prototype implementation of the system presented and discussed in the previous chapter is necessary to gather user's feedback and subsequent improvement to the design. Although a complete functional implementation of the Integrated Training and Decision Support (IT-DS) system is beyond the scope of this project, parts of a complete prototype can contribute towards the understanding of the final system. This chapter begins with a general discussion of the implementation issues that will face this partial effort that also includes presentations and rationales for the approaches chosen. Next, the principal components of the prototype system built for this work will be presented. These include the database component and the graphical user interface component.

5.1 Implementation issues

The implementation of a prototype based on the specification models discussed in **Chapter 4** needs additional decisions involving such issues as the approach to use in building the components and objects that make up the system; the availability of suitable technology in database support, in building distributed simulation engine and distributed knowledge bases (for expert system), and in building the user interface. The topics of database support and building a user interface are the focus of the prototype discussed in this work,

whereas the additional work of integrating expert systems and simulation are subjects worthy of their own theses and are thus beyond the scope of this project.

5.1.1 Software development approach

There are currently two general approaches available for software development: using a single language or using components. The single language approach involves using a language such as C++, Ada or Java to build the application using customized codes and using same language libraries that have been constructed by others. The advantages of this approach include semantic uniformity and the ability to take advantage of a particular language feature, power and speed. However, the advantages are also disadvantages, such that if libraries are not available, codes then have to be built from scratch and the potential for over-extending the language, using it in areas where it is weak. On the other hand, one has the component approach that is popularized by Microsoft Visual Basic (VB) development platform. Using VB, developer can effectively build application by assembling together different components built using then the Visual Basic extension (VBX), and OLE[1] component extension (OCX) technologies. The component approach is much more powerful and flexible than the single language approach because components can be built using the most suitable language and they are also interchangeable, yielding better maintenance and performance advantage.

---

[1] OLE originally stands for Object Linking and Embedding. However, Microsoft has discouraged developers from using OLE in this sense.

It is clear that it is advantageous that the prototype is developed using a component approach, so that other features such as expert system and simulation integration can be added readily when they will become available. However, a component technology and its associated architecture must be selected before the work can begin. A component technology and architecture represent the supports to mediate communication among components. There are currently two such technologies: the Component Object Model (COM) from Microsoft and Java Beans from Sun Microsystems. Java Beans is an emerging technology and is not yet as mature as COM which is supported by many tools from Microsoft and other third party vendors. However, Java Bean development is currently only available using the Java language, whereas COM is designed to be language neutral and can be developed using C++, Fortran and Java. Nevertheless, Java Beans is promising as a component architecture that will take advantage of the cross-platform feature of the Java programming language and system compared to COM which currently only works on the 32-bit family of Windows operating system.

For the prototype purpose of this work, one needs two kinds of tools: one for the user interface part and another for the construction of the persistence mechanism for the classes of the domain objects modeled previously. COM will be chosen for its maturity in term of development tools, and as a COM enable tool, Visual Basic will be used for user interface development (see **Appendix B**

for a brief introduction to COM and the COM Automation development process). However, Java will be chosen as the development language for the actual persistent component with COM tools from Microsoft (Visual J++), a reason that will be discussed further in the next section. The issue of whether to migrate to Java Beans for cross-platform deployment is not considered at this stage of the development; however, tools are emerging for this task if there is a need for it. Using Java as the development language also helps in this future requirement.

5.1.2 Persistence technology

Data that are collected by the operator, data about the users of the system, application dynamic logic (*e.g.*, calculation formula script) need to exist beyond the execution moment for permanent record keeping and for fault tolerance purpose. The solutions to this problem include flat file storage, hierarchical storage or using a specialized database management system (DBMS). For simple application and relatively unchanging data, a simple flat file or hierarchical solution suffice. However, for complex application that tends to evolve constantly, using a DBMS simplifies development tasks and provides convenient maintenance and operational mechanisms. There are currently three data models that are the driving forces behind modern DBMS technology: the relational model, the object oriented model and the hybrid object-relational model. Database technology that uses the relational model is commonly referred

to as Relational DBMS (RDBMS); Object Oriented DBMS (OODBMS) uses the object oriented model while Object Relational DBMS (ORDBMS) uses the object-relational model.

Choosing one among these technologies is a considerably complex task that would take into account several factors such as the usage pattern of data in an application, level performance required, budget and technological expertise on each technology to name a few. Each database technology has its own advantages and disadvantages and technical merits (see Cattell, 1994 and Rao, 1994). However, it is commonly acceptable to use OODBMS as a rule of thumb when dealing with complex data, when using an OO design methodology and when maximal flexibility is required (Cattell, 1994). The prime advantage of using OODBMS is the ability to make objects persistent directly without using translation code that is often tedious and error prone.

For the prototype version of this project, Object Store PSE for Java, supplied by Object Design Inc. will be used. This is a small footprint OODBMS that is freely available for non-commercial use. This object oriented persistent engine for Java has the same interface as the full feature Object Store product from the same company, thus guarantees future expansion and easy scale-up work after the prototype phase. PSE has interfaces for creating, opening, and destroying databases. It has full support for transaction and can store up to 100 MB of data without performance degradation. PSE implements persistence by

reachability and automatically save changed object within a transaction. However, PSE lacks features such as support for multiple read/write transaction, automatic recovery, lock detection and multiple sessions that are features of the Object Store product.

5.1.3 User Interface technology

The IT-DS system should be accessible to operator using the most convenient interface possible, which may include rugged touch-screen that can be used in industrial setting, voice or feature key activation. The proper design of the user interface should be the primary concern after the functional components of the system have been built and tested. For the prototype purpose, a simple graphical user interface (GUI) using the Microsoft Windows environment suffices. The design and implementation of this GUI could have been done in pure Java or using tools such as the Microsoft Foundation Classes (MFC) and Visual Basic. Again, Visual Basic is chosen for its relative maturity and richness over GUI development in Java, which is still evolving. In the future, the cross-platform and cross-device feature of Java development may be considered in designing and implementing user interface for use in industrial setting, but only when sufficient tools have been developed.

5.2 The ASDB Component

The design for the first version of the ASDB Component as discussed in **Section 4.3.2** (**Figure 4-18**) exposed two interfaces: IManageData and IQuery. It

is these two interfaces of methods that will make up the Interface Definition Language file (IDL) (**Appendix B**). The IDL file is used by COM tools such as the Microsoft Interface Definition Language (MIDL) compiler to create the type library that will be used by Visual Basic to call the methods and set the ASDB Component properties. Internally, these methods are implemented in Java, within the ASDB class. This class encapsulates two helper classes: QueryManager and DataManager that will interact with the actual OODBMS application programming interface (API) of the PSE engine.

In the following sections the semantic of each method of the interfaces IManageData and IQuery will be presented (Note: method declarations are written using Java syntax). These interfaces treat the data objects as encoded/decoded string, a necessary work around, since the current Microsoft COM implementation for Java does not work completely with the PSE engine[2] (more details later). Following COM convention, each method can also return an integer for error code.

int OpenDB(String _target, int _status)—this method takes two parameters: the name string of the database and the status code. It then initializes the PSE engine, attempts to open the named database and set it to the status code. Upon

[2] Object Design is well aware of this temporary incompatible problem, and has made a number of patch releases. However, the patches are not stable enough to risk more time on various work-arounds. However, this is not a problem because COM interfaces can be extended to include new and stable solutions without affecting existing users.

succeeding, the method will return an integer that is the *id* for subsequent communication, otherwise it returns an integer error code.

int CloseDB(int _dbID)— this method attempts to close the opened database identified by *_dbID*. It will return 0 if it succeeded and other error codes if not.

int SelectDB(String _target)— this method finds the database *id* of the internal database table based on the name received. Upon succeeding, it return the id, otherwise it returns an error code. This method is useful in multiple database scenarios.

int CreateDB(String _target) — this method creates the specified database and set it to receive update. Upon succeeding, it returns the database *id*, otherwise it returns an error code.

int InsertObject(int _db, _Otype String, String[] _parArray) — this method inserts a new object of type *_Otype* into database *_db* with object attributes encoded as an array of String. It returns 0 when it succeeds and an error code when it does not. *_Otype* is a string that identifies the type of the object. For example *_Otype* can be *Observation, Measurement* or *HTMLDoc*. These type-identifying strings are used inside the ASDB Component to decode the attribute string into attributes. Then object of type *_Otype* are created and stored in PSE database. Due to the current limitation of COM support of the PSE engine, a direct object insertion is not possible. However, this approach of encoding an

object may also be beneficial if at a later date, a RDBMS is used instead of an OODBMS.

int UpdateObject(int _db, _Otype String, String[] _oldparArray, String[] _newparArray)— this method update the object of type *_Otype* in database *_db* by comparing the old and new attribute arrays. It returns 0 for success and an error code otherwise.

int DeleteObject(int _db, _Otype String, _parArray String[])— this method deletes the object of type *_Otype* from the database *_db*, given the attribute array *_parArray*. It returns 0 for success and an error code otherwise.

String[] GetOneObject(int _db, String _Otype, String[] _parArray) —this method find the object of type *_Otype* in database *_db* and returns the array of String objets that encodes the object states. It will return null if it does not find the object. This method assumes that the database identifier is valid.

int InitGetManyObject(int _db, String _Otype, String[] _start, String[] _stop)— this method initializes the internal mechanism for getting several objects of type *_Otype* in database *_db*, with starting states matching the array of *_start* and ending with those matching the array *_stop*. It returns 0 for success and an error code otherwise. After a successful initialization, the client can use the method *GetNextObject()* to retrieve the objects' encoded strings.

int InitGetLogic (int _db, String _Otype, String _op, String _cond1, String _cond2)— this method find the objects of type *_Otype* in database *_db* using operator AND, OR or XOR passed in *_op*, with operands *_cond1* and *_cond2*. It returns 0 for success and an error code otherwise. The client use the GetNextObject() method to get the found objects.

int InitGetLogicNot(_db int, String _Otype, String _cond) — this method find all the object of type _Otype in database _db that does NOT satisfy the condition _cond. It returns 0 for success and an error code otherwise. The client uses the GetNextObject() method to get the found objects.

String[] GetNextObject()— this method is called only after the appropriate initialization and returns one String array after the other until it returns null that signifies no more object available.

The ASDB Component as currently implemented is not very efficient since a significant amount of effort is spent on encoding and decoding the various objects that need to be stored or retrieved from the database. Furthermore, objects that have behaviors such as *ASLudge* and *Plant* (see **Chapter 4**) also have to expose these methods if they are to be used by the VB client. These method exposures are not yet available due to the evolving nature of PSE. However, these limitations are easy to overcome once the tools from ODI mature, which is forthcoming at the time of this writing. One may argue that such risks should have been foreseen and a more traditional approach such as using a RDBMS to

store data and perform the final processing at the client is more suitable. However, even with the temporary setback because of tool immaturity from the part of ODI, the effects are acceptable from the point of view of design, which is the focus of this thesis. Having the database engine dealing with code and data together, as whole object is much more manageable and maintainable in the long run than relying too much on the largely procedural approach available in Visual Basic.

5.3 The graphical user interface prototype

The inclusion of a graphical user interface (GUI) to computing systems has become the norm rather than the exception nowadays. In order to gradually refine the interface and present the functionality of the system a prototype GUI system built using Visual Basic 5.0 (VB5) has been constructed. VB5 provides the fastest way to create GUI prototype system based on the MFC library for the Windows operating systems. It can also use COM component such as the ASDB Component discussed previously.

The list of features for the interface normally reflects user wishes and available functionality of the underlined system. For the time being, the IT-DS system is by no mean complete such that all features can be developed. Although the prototype includes placeholders for some of these features, they are not all yet accessible. The feature set currently available focus on the ability to create, store and retrieve water records and documents. Some of the various

routinely used calculation procedures such as Return activated sludge (RAS), Mean Cell Retention Time (MCRT) are available; however, these are not yet changeable by the user.

A base multiple document interface (MDI) application was first generated using the VB5 application Wizard. Subsequent development adds to the generated application skeleton. Aside from the standard menu generated by the Wizard, additional menus items and buttons were added to serve the two category of user of the system: the administrator and the members of the administrator group; the normal user and the user group (*i.e.*, plant operator). As user is logged on to the system, the user identification will be used to turn on or off certain functions.

A full feature GUI prototype serves the following functions to the administrator:

1. Create and add a new user to the database
2. Set up plant information
3. Set up database
4. Set up simulation models
5. Set up knowledge bases
6. Set up documents

For the normal user, the plant operator it serves the following functions:

1. Create/edit/update water record and associated observations and measurements.
2. Create/edit/update case-log and associate information
3. Create/edit/update meta-data concerning case-log and water record
4. Generate various reports and charts
5. Manage user preferences and personal profile

For the partial prototype of this work, the administrator can not set up simulation models or knowledge bases, and the user can not generate reports, charts or create personal preferences and case-log. These features will be eventually available in future prototypes.

5.4 Description Current Prototype User Interface Features

After the start-up phase, the prototype will ask the user to log-in the system (**Figure 5 - 1**). The additional menus that have been added to the standard MDI menus were **Logs**, **Reports** and **Tools** (**Figure 5 - 2**). The administrator of the system manages the system configuration through the **Tools** menu, which includes functions to manage users, set up plant data, database, simulation models and knowledge bases (**Figure 5 - 2**). It is also through this menu that the normal user (operator) would access the **Search** and **Browse** features to work with stored data and also the **Simulation** and **Diagnostics** functions (future features).

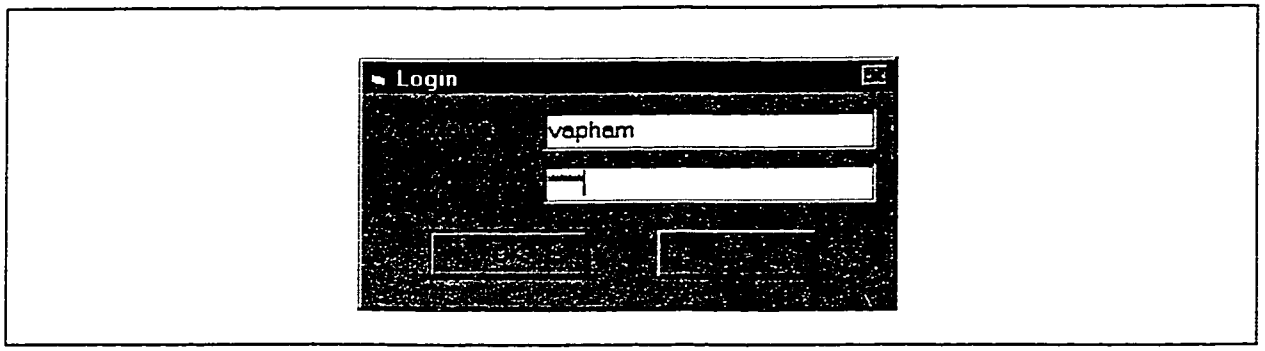


Figure 5 - 1 IT-DS System Login

Data entry to the system such new water record, case-log, observation and measurement would be available to the operator *via* the **Logs** menu (**Figure 5 - 3**). The other menu of interest is the **Reports** menu, where it is only a place holder for now, but it will be through here that the operator would produce plant and process summary report, and would see various charts describing the plant's operation states produced (**Figure 5 - 4**).

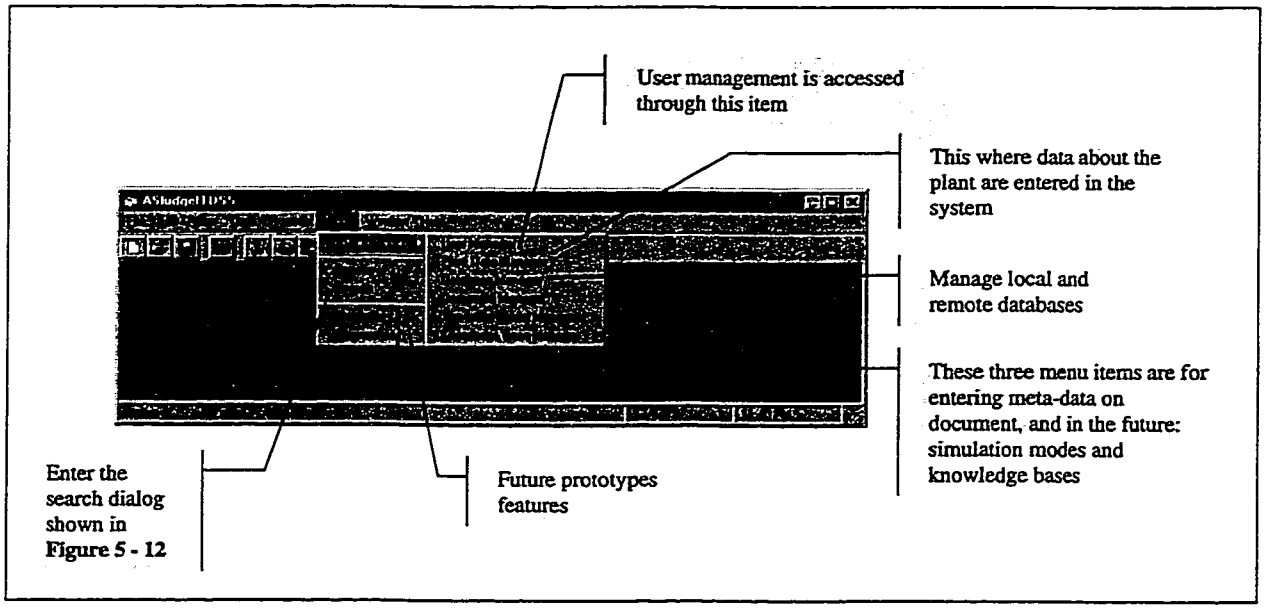


Figure 5 - 2 The Tools menu

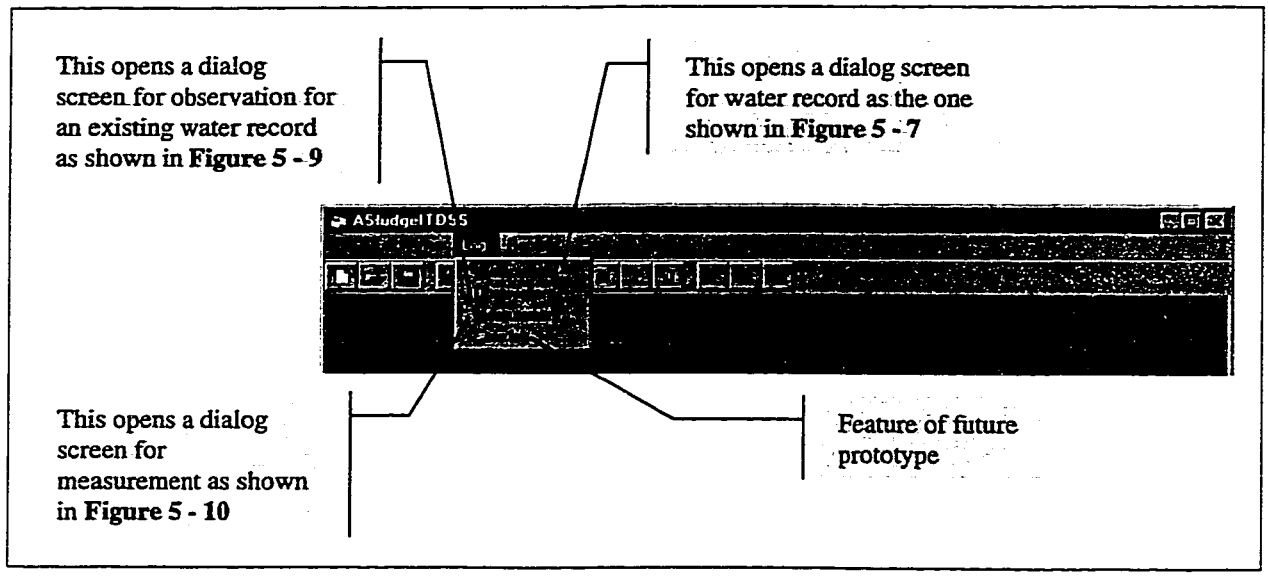


Figure 5 - 3 The Logs menu

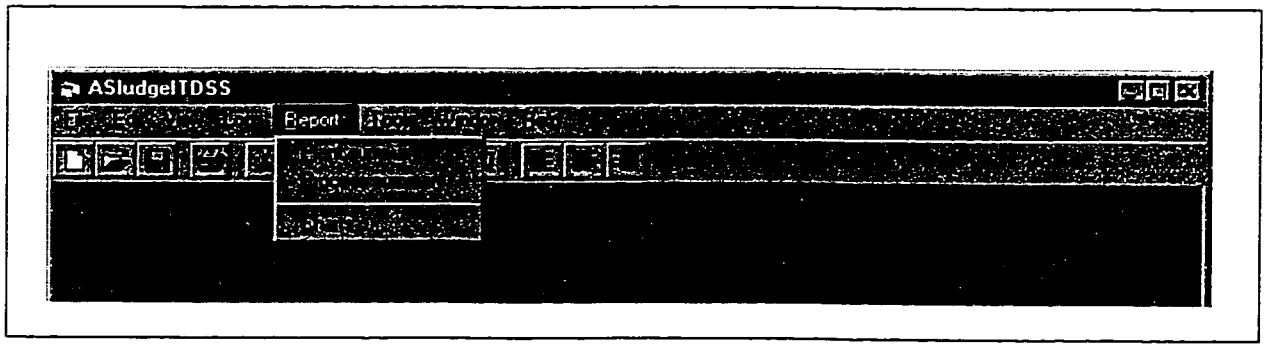


Figure 5 - 4 The Reports menu

After the ASDB component is created, its features are accessed from Visual Basic 5.0 *via* the interface (methods) described previously. The ASDB is asked to store and retrieve objects that encapsulate User data, Water record, Observation, Measurement and Meta-data after the user interface processes the data and encodes them into a long string. This encoded string is passed over into the ASDB component where the long string is convert back into data based on the type of object and stored into the database. The process is reversed when

data is retrieved. This encode-decode step is the current bottleneck and the weak point of the system; the extra step is necessary because of the current limited support of COM for Java. However, it is no longer needed when a direct interface between PSE and COM is available. Despite this shortcoming, what matters to the end user through the user interface is largely unaffected. In the following sections, the highlights of the graphical user interface will be presented.

To begin, **Figure 5 - 5** shows the interface that the administrator adds, deletes user to and from the system and assigns user privileges. The administrator performs these tasks through the **User** and **Policies** menus. The user information is represented by two lists denoting the individual accounts and the groups that are available.

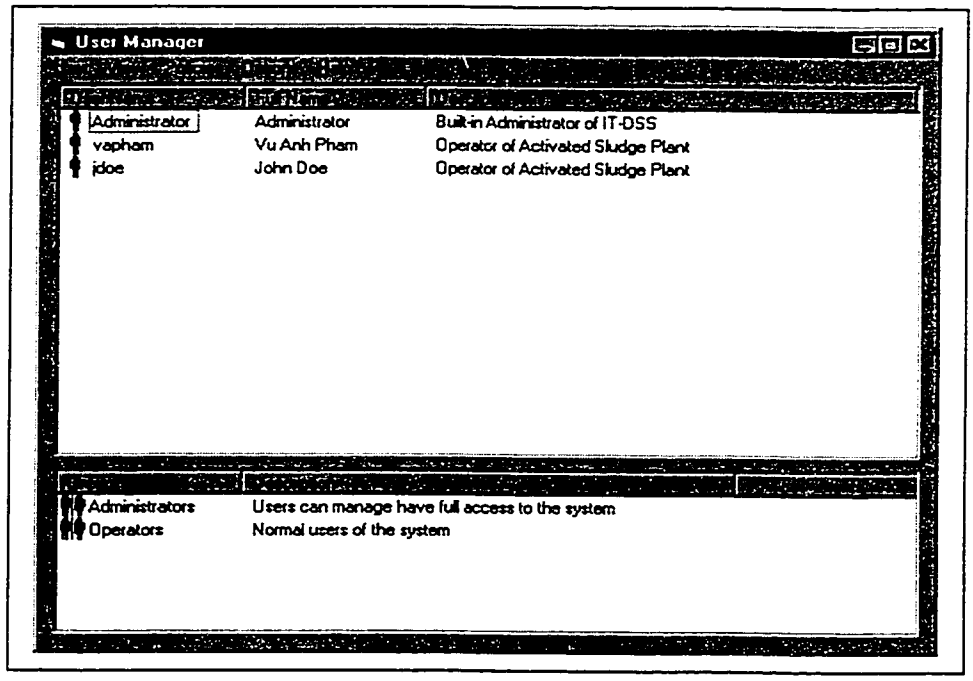


Figure 5 - 5 The user manager interface

Once the user is added to the system, s/he can start working after logging in. The user can interact with the various features *via* pop-up menus that are available from the plant diagram (**Figure 5 - 6**). If a new water record is being created, an interface as shown in **Figure 5 - 7** is presented to the user.

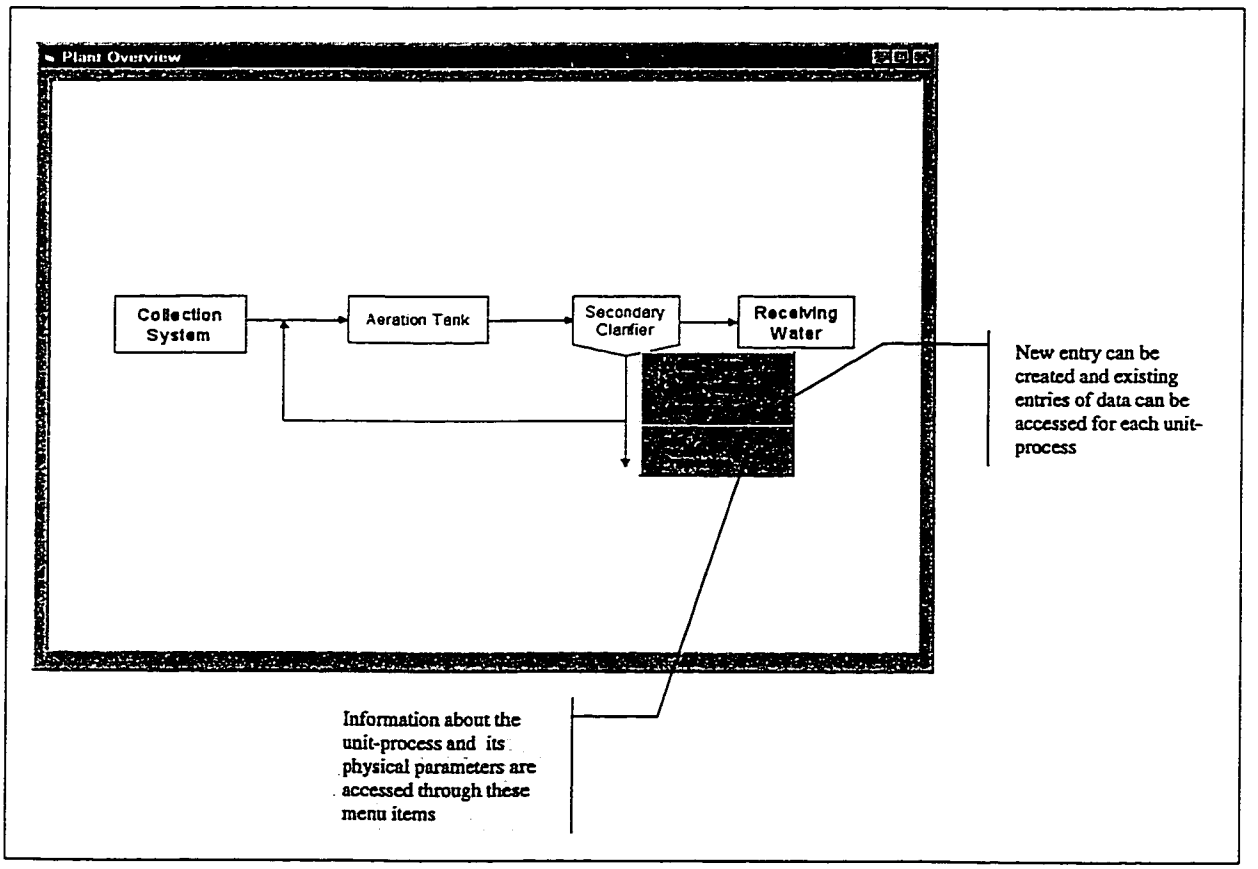


Figure 5 - 6 Pop-up menu from plant overview

From here, the user can create new water record, observation and measurement by clicking on the buttons on the tool bar. Observations and measurements have to be associated with a water record, and are shown for each water record entry in the lower view as shown in **Figure 5 - 7**. When a new

water record is being created, the user also has the option to enter associated observations or measurements directly from inside as shown in **Figure 5 - 8**. There are also navigational features (**Back** and **Next** buttons) to facilitate multiple entries.

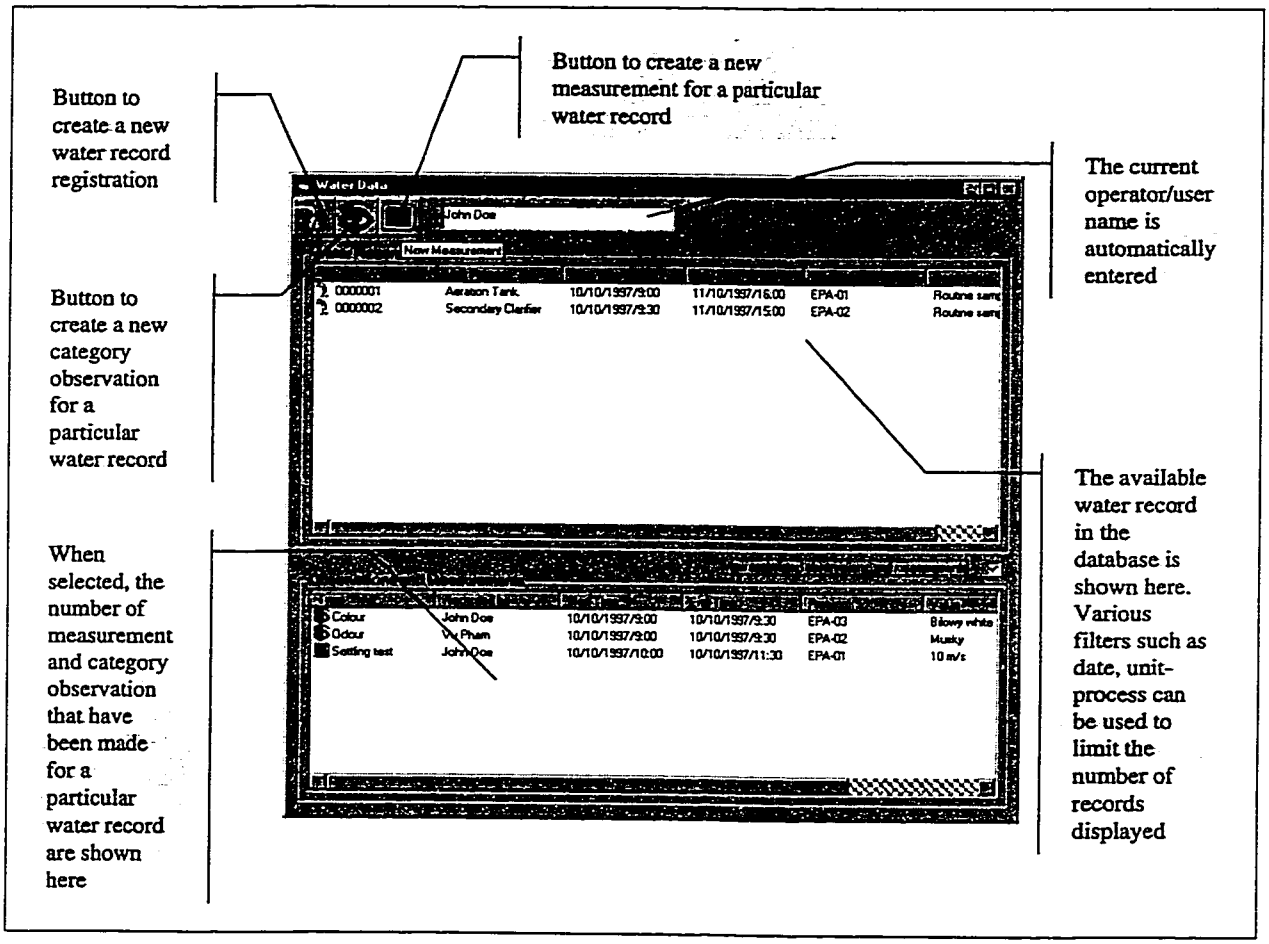


Figure 5 - 7 Water record data interface

The interface for entering observations and measurements are self-explanatory and are shown in **Figures 5 - 9** and **5 - 10** respectively. In all the interfaces presented so far, information such as the user identification and unit of measurement are filled automatically. Information that is necessary for the meta-

data are also being extracted automatically (author extracted from the operator field, keywords generated by processing the content of the description area and the description for the abstract itself (see **Section 4.3.5**). These and similar data in case of simulation or diagnostic can be automatically generated because they can be retrieved from the database (future features).

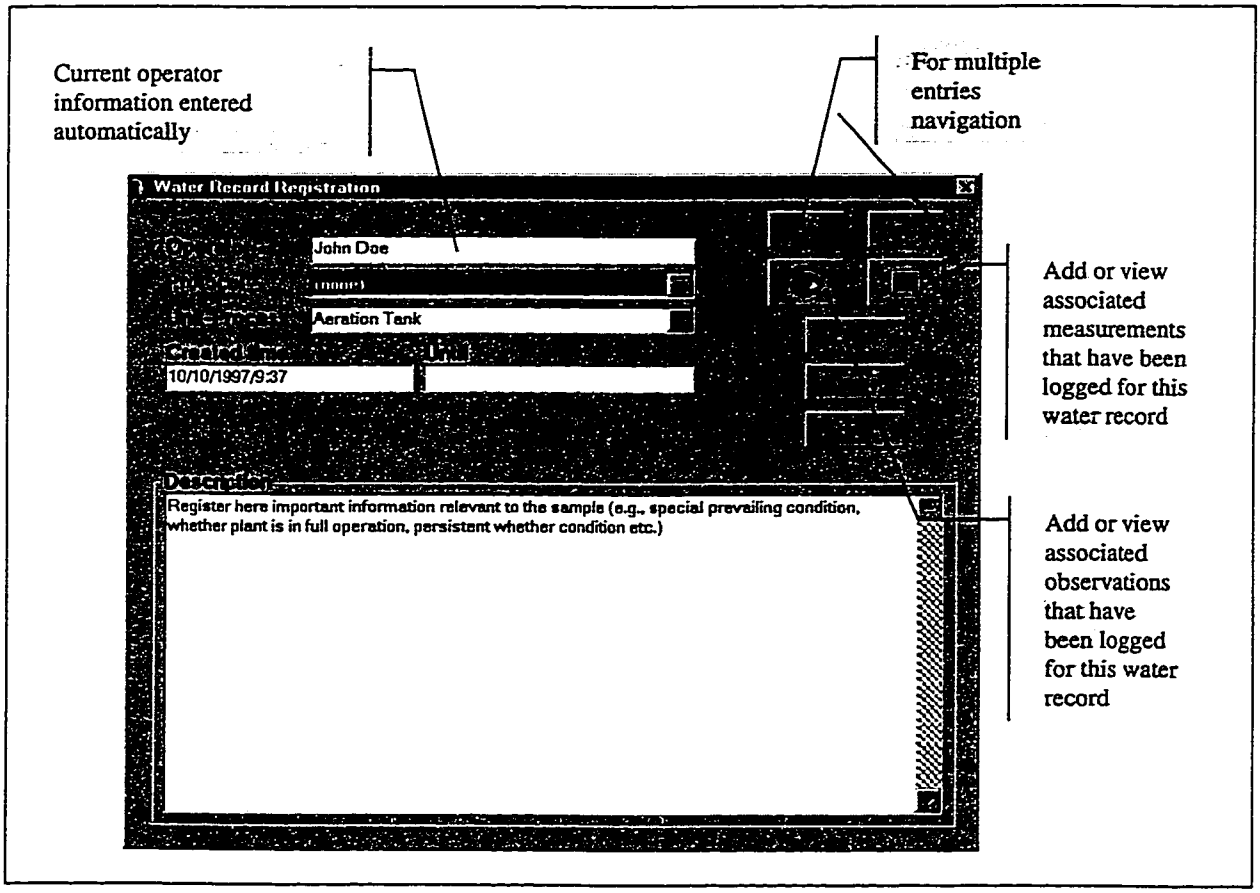


Figure 5 - 8 Interface to enter water record

The meta-data about documents, simulation models, knowledge bases can be entered into the system *via* an interface shown in **Figure 5 - 11**. Four areas of information have to enter: author statistics, time statistic, description and keywords. An option to have the system automatically generates suggestions to

the above fields from existing data is also shown but is not yet implemented. Once completed, these are the information that the system uses in searching for the needed information and helping the user decide on appropriate action to take regarding which simulation model or knowledge base to use, or which data range should be included in the calculation.

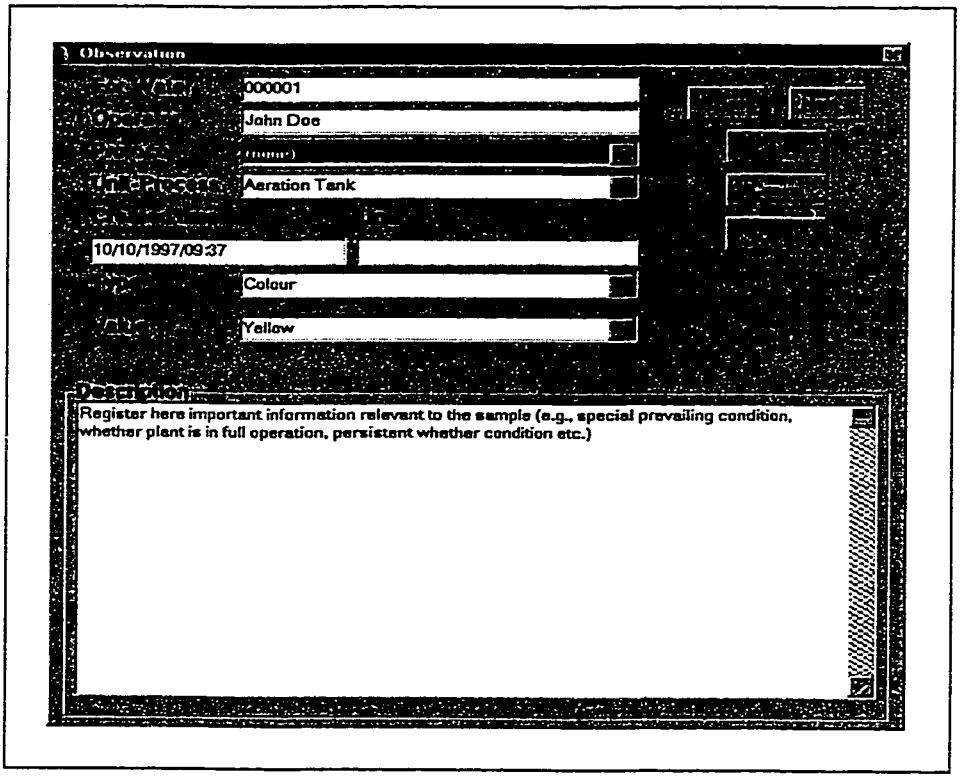


Figure 5 - 9 Interface for new observation

Once the data have been available in the various databases, the user can initiate a search for range of data with an interface shown in **Figure 5 - 12.** In this first prototype, the user can enter keywords and choose the Boolean operator for the search expression. The user can also limit the number of results to return. Moreover, s/he can also choose what to collection of data to include in the search as well as the temporal limitation by specifying the applicable starting and

ending time point. The system also keeps a set of most recent search results as a list in the **Recent** tab shown in **Figure 5 - 12.** From typical search results (**Figure 5 - 13**), document can be accessed (and example is shown in **Figure 5 - 14**), simulation model can be launched, knowledge base can be initialized and accessed, water records and associated observations and measurements and be examined.

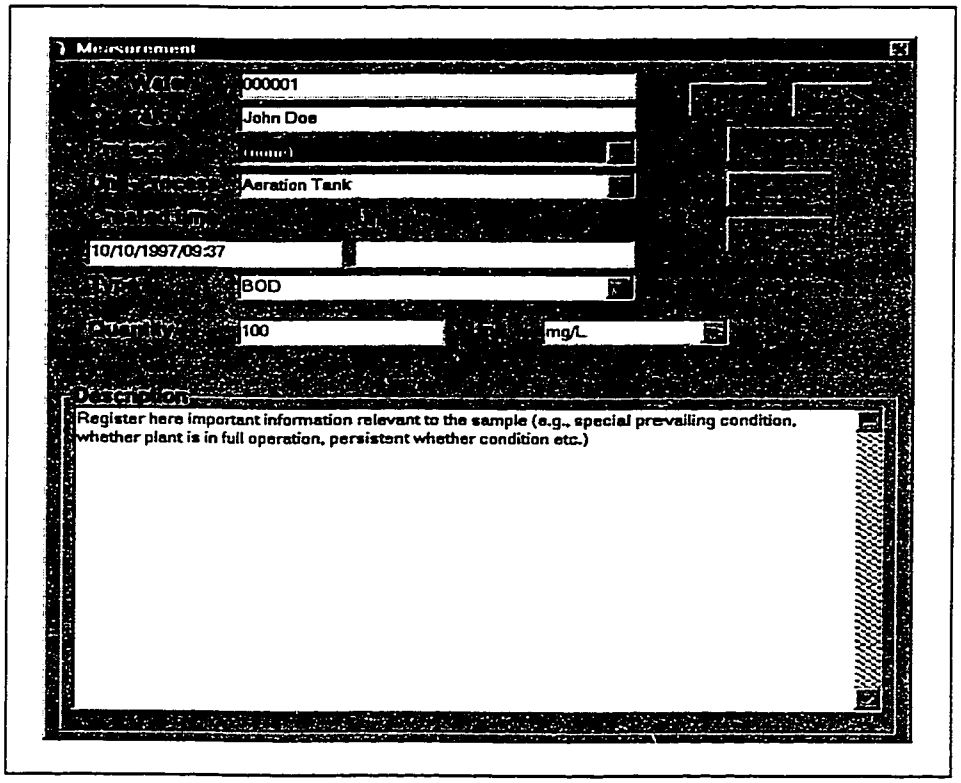


Figure 5 - 10 Interface for new measurement

The search interface as all other presented in this part of the thesis could be further improved to have many more functionalities that truly reflect all the richness of the object models that they provide access to. However, due to unanticipated constraints, their development have to be stopped at the current

state, which is unfortunate. However, for the purpose of the implementation of the prototype, which is to show how the different pieces work together, the interfaces presented have met their objectives.

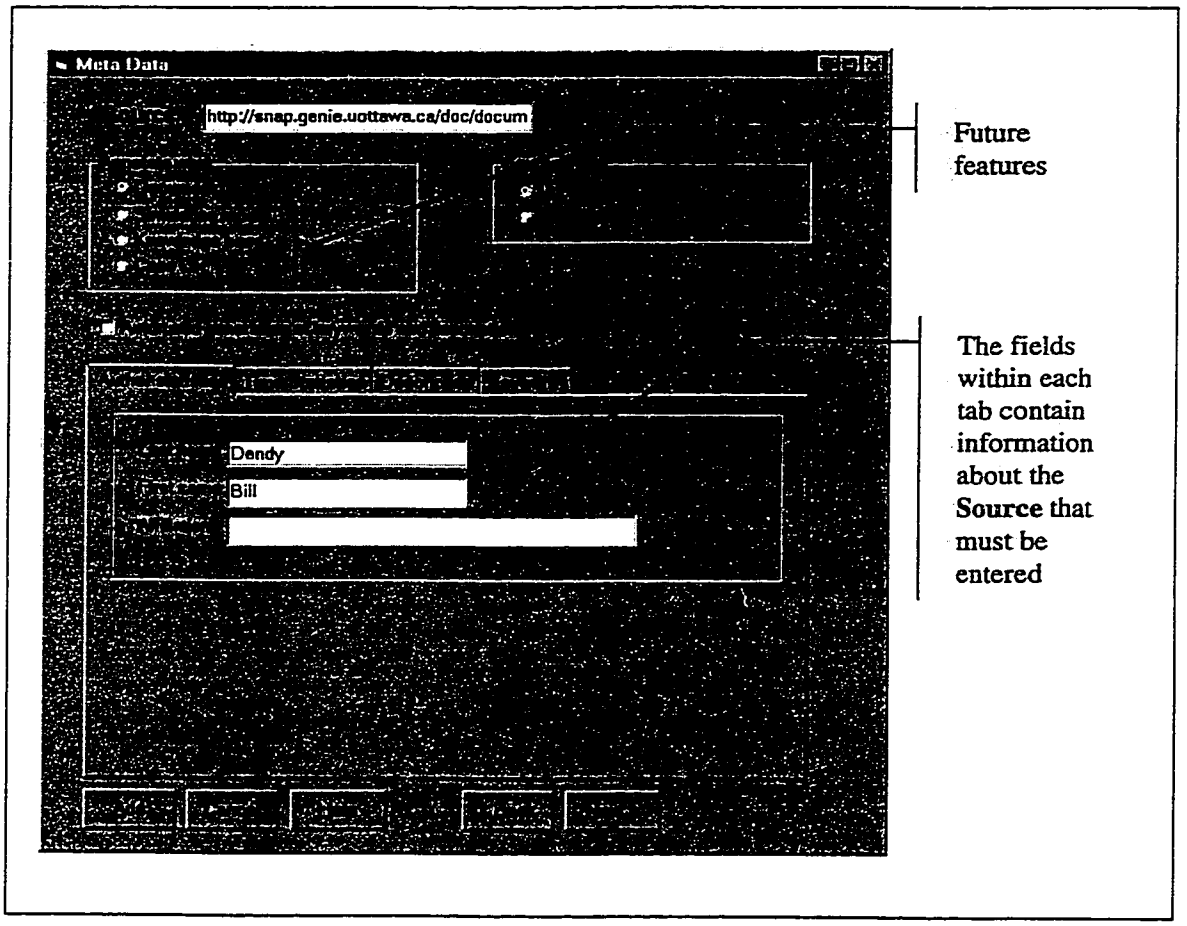


Figure 5 - 11 The meta-data interface

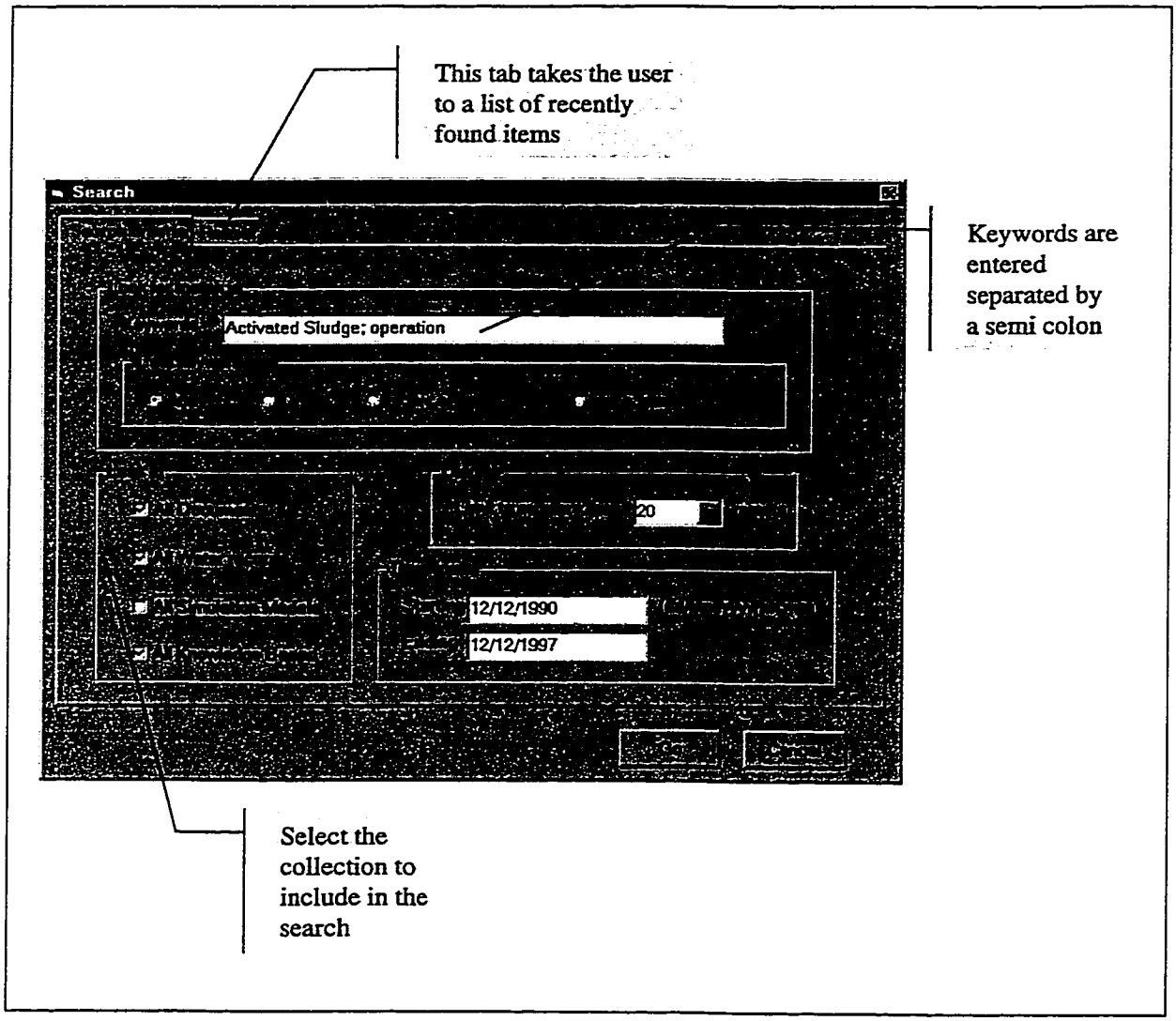


Figure 5 - 12 The search interface

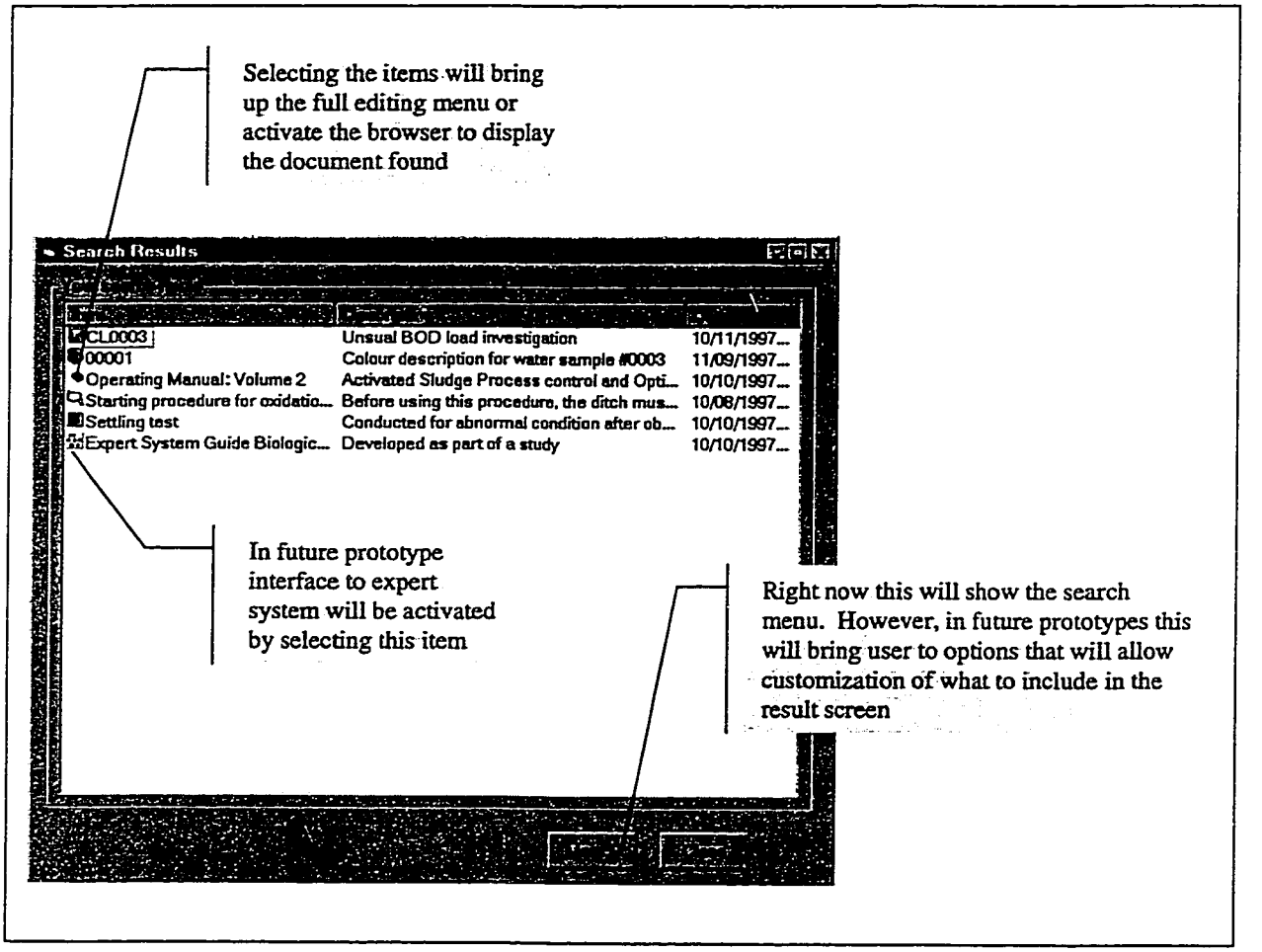


Figure 5 - 13 Typical search results

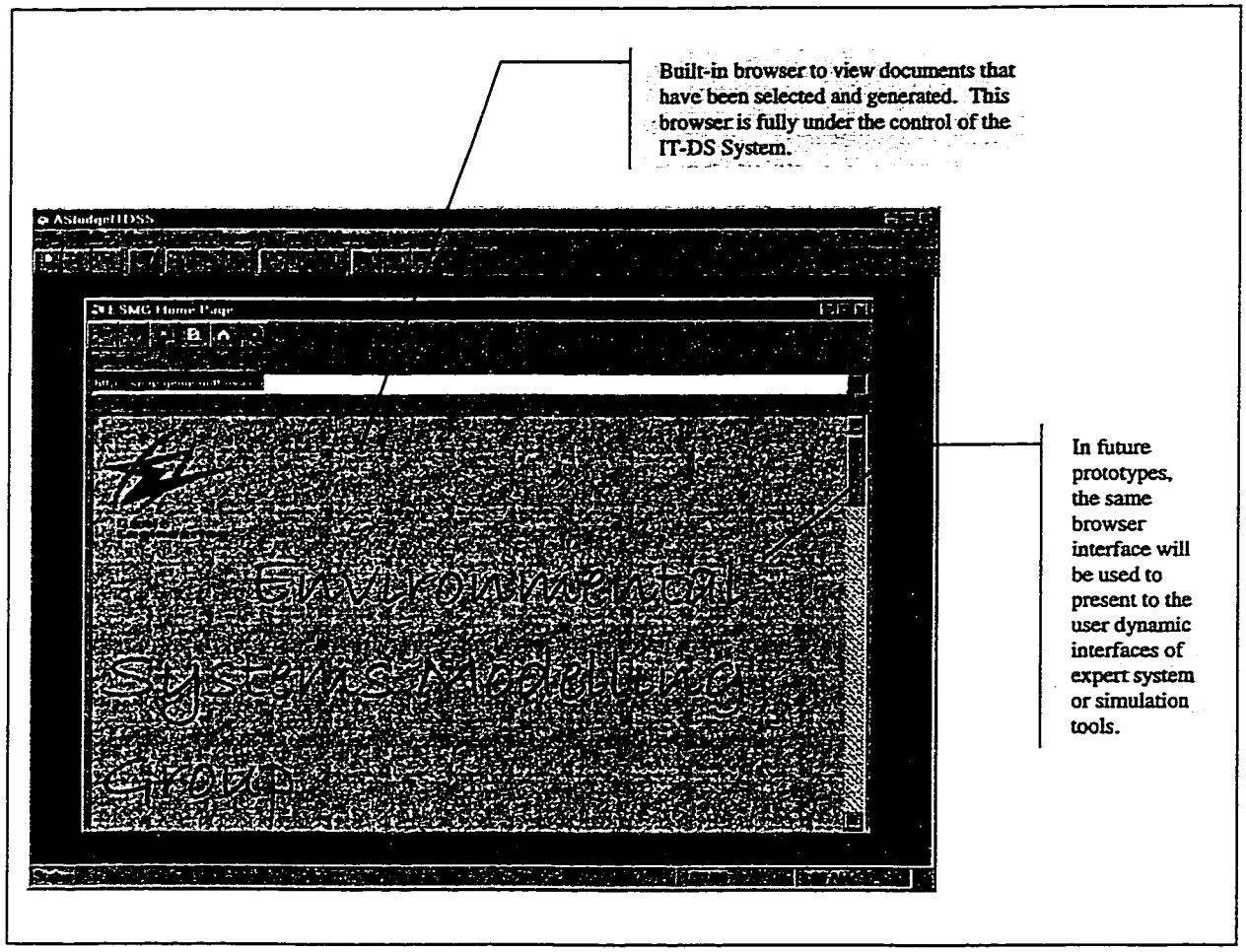


Figure 5 - 14 Using the programmable browser to display document

Chapter 6

CONCLUSIONS AND RECOMMENDATIONS

This thesis was set out to integrate the data collection task and the various tools, that an operator at a wastewater treatment plant may use, into an Integrated Training and Decision Support System (IT-DSS). The project has been motivated by observing the shortcomings of current tools in use at plants, such as information management system, simulation models and expert systems, as discussed in Chapter 1 and 2. The idea was to allow greater data reuse and sharing among several systems and to present a unified interface to the user. To this end the requirements of the system have been researched and developed. Object- oriented software development methodology was discussed and selected, and various object models have been synthesized. A prototype with limited functionality has also been developed.

As discussed previously, the development of a fully functional system is clearly beyond the scope of a master project. This is unfortunate, because a fully functional prototype would facilitate the gathering of feedback from the operator on the proposed system, so that it can be further evaluated and improved. However, this thesis has shown that the design of the IT-DS System is feasible and that the system is realisable. Furthermore, all four goals that have been

enumerated in Chapter 1 have been completed satisfactorily. Again, these goals were:

1. Evaluate the feasibility of designing the structure of the IT-DSS.
2. Evaluate whether the system is realisable.
3. Synthesise conceptual and specification designs.
4. Evaluate the system design *via* a prototype implementation.

Evaluating whether a software project is feasible and realisable is an important milestone to achieve. This is important because it allows the proper management of resources and risks, factors that are crucial for the successful completion of any software system. The IT-DS System is feasible since its design is based on sound design principles, and it is realisable because the implementation technologies chosen are well known.

Appropriate and efficient software design is a difficult endeavour and the synthesised conceptual and specification models that have been presented speak volume to the usefulness of design patterns and object-oriented methods. Without these tools, the system models may have taken longer to complete, may have needed more refinements and may not be reusable. However, design patterns and object-oriented methods are not silver bullets. Their uses only promise the designer more room to manoeuvre if things went wrong. They are useless to weather the technological risks that may be encountered in implementing the system.

Overall, the outlined work has helped to achieve many objectives. These achieved objectives are very important and they represent the completion of the first phase of the project. It has been demonstrated *via* the models obtained and the partial prototype construction that the integration concept has practical appeal and could be designed and implemented. A reusable software development approach using software components was proposed. It was intended to facilitate the development and future enhancement of features such as the replacement of the Microsoft Windows based interface with a more industrial oriented one like touch-screen, that can be designed to work in the rough environments of a wastewater treatment plant.

For future studies, the risk and impact of technological immaturity should be particularly noted. During this study several features of the object models thought to be available in the Java language were in fact not yet available. Therefore, the prior decision to take advantage of the language features such as automatic memory management and simpler syntax can not be fully exploited. It is unlikely that these features will be readily available in the near future. Therefore, the next version of the prototype should be implemented using a more mature language such as C++.

However, experimenting with Java should not be eliminated completely because Java provides a simpler mechanism for dynamic interface delivery that could facilitate the changes in simulation models functions and knowledge bases.

Java applets could provide the user with a uniform and dynamic interface to these tools (simulation models and knowledge bases) *via* the browser interface as proposed with the meta-data facility. This would allow developers of simulation models and knowledge bases to flexibly change the interface to their tools, while using the meta-data proposed in this work to maintain user access.

The distributed computing model of the Internet, enabled by the popularity of the World Wide Web browser and related technologies (*e.g.*, HTTP protocol, Java and ActiveX) creates a new flexible design option that simply did not exist before. However, Internet development and integration also suffer from immature product features or lacking there of that can make this project very difficult to manage. Therefore, a fully functional prototype is anticipated only after certain features of the development environment have stabilized somewhat and are robust enough so that a good prototype can be built and used to work with the end user.

Software tools such as document converter or extractor can also be made available to document developers as well. This will simplify the process of making documents available to users, as well as the ability of users to change the actual documents. One possibility is to work on integrating various authoring tools to allow saving of information such as meta-data directly while authoring. Another additional feature to be considered is the integration of the IT-DS system with the existing SQL database that may be available at the plant to

provide added benefits. This task is not difficult since this thesis work used the Visual Basic environment. However, after these additional works have been carried out, real experimentation to determine the usefulness of the system will have to be conducted. Only then can a well-rounded assessment of the IT-DS system be obtained and the thoughts and ideas proposed by this thesis be truly tested and validated.

REFERENCES

Alexander, C., *The timeless way of building*, Oxford University Press, 1979.

AllMax Professional Services Corporation, *Operator 10*, AllMax, 1996.

André, E. and Rist, T., "Towards a plan-based synthesis of illustrated documents", *Proceedings of the 9th European Conference on Artificial Intelligence*, 25-30 (1990).

Andrews, J.F., "Dynamics and Control of Wastewater Treatment Plants", *Proceedings of the conference on Computers in the water environment*, August 8-11, 1993; Santa Clara, California; Water Environment Federation.

Barnett, M.W. and Andrews, J.F., "Expert System for Anaerobic-Digestion-Process Operation", *J. of Environmental Engineering*, **118** (6), 949-963 (1992).

Barnett, M.W. and Andrews, J.F., "Knowledge Based Systems for Operation of Wastewater Treatment Processes", Conference on Instrumentation Control and Automation of Water and Wastewater Treatment and Transport System, *Proceedings of the 5th IAWPRC Workshop*, Yokohama, Kyoto, Japan, July 26 - August 3, 1990. Ed. Briggs, R., City University, London, UK.

Beck, K. and Cunningham, " A Laboratory for Teaching Object-Oriented Thinking" in *Proceedings of OOPSLA 89*, SIGPLAN Notices, **24** (10), 1-6 (1989).

Booch, G., *Object solutions : managing the object-oriented project*, Addison-Wesley, 1996.

Booch, G., *Object-Oriented Anaysis and Design with Applications*, Benjamin/Cummings, 1994.

Boy, G.A. and Mathe, N., "Operator assistant systems. An experimental approach using a telerobotics application", *Int. J. of Intelligent Systems*, 8 (2), 387-396 (1993).

Brockschmidt, G., *Inside OLE*, Microsoft Press, 1996.

Burnett R. E., *Technical Communication*, 2nd Edition, Wadsworth, 1990.

Burns H., Parlett J. W. and Redfield C. L., eds., *Intelligent Tutoring Systems: Evolutions in Design*, Lawrence Erlbaum Associates, New Jersey, 1991.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M., *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons, 1996.

California State University, Sacramento, Department of Civil Engineering, *Operation of Wastewater Treatment Plant*, 4th Edition, Vols 1 & 2, Environmental Protection Agency, 1996.

Cattell, R.G.G., *Object Data Management*, Revised edition, Addison-Wesley, 1994.

Coad, P. and Mayfield, M., *Java Design: Building Better Apps & Applets*, Yourdon Press Computing Series, 1996.

Coad, P. and Yourdon, E., *Object-Oriented Analysis*, Yourdon Press, 1991.

Coad, P., "Object-Oriented Patterns", *Communications of the ACM*, **35** (9), 152-159, 1992.

Coad, P., North, D. and Mayfield, M., *Object models : strategies, patterns, and applications*, 2nd Edition, Yourdon Press, 1997.

Cook, S. and Daniels, J., *Designing Object Systems: Object-Oriented Modeling win Syntropy*, Prentice Hall, 1994.

Coplien, J.O. and Schmidt, D.C., Eds., *Pattern languages of program design*, Addison-Wesley, Reading, 1995.

Davis, A. M., *201 principles of software development*, McGraw-Hill, 1995.

Firesmith, D. G., *Object-oriented requirements analysis and logical design : a software engineering approach*, Wiley, 1993.

Fowler, M. and Scott, K., *UML Distilled: Applying The Standard Object Modeling Language*, Addison-Wesley Object Technology Series, 1997.

Fowler, M., *Analysis Patterns - Reusable Object Models*, Addison Wesley, 1997.

Frasson C. and Gauthier G., eds., *Intelligent Tutoring Systems: At the Crossroads of Artificial Intelligence and Education*, Ablex Publishing, New Jersey, 1990.

Gall, R.A. and Patry, G.G., "Knowledge-based system for the diagnosis of an Activated Sludge Plant", in *Dynamic Modeling and Expert Systems in Wastewater Engineering*, Patry G.G. and Chapman D. Eds., Lewis Publishers, Chelsea, MI, 1989.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995.

Garrity, E.J. and Sipior, J.C., "Multimedia as a vehicle for knowledge modeling in expert systems", *Expert Systems with Applications*, 7 (3), 397-406 (1994).

Goodman, B., "Multimedia explanations for intelligent training systems", *Proceedings of the Conference on Intelligent Computer Aided Training*, Houston, TX, 139-153 (1991).

Jacobson, I., Ericsson, M. and Jacobson, A., *The Object Advantage: Business Process Reengineering with Object Technology*, Addison-Wesley, 1995.

Jacobson, I., *Object-Oriented Software Engineering: A Use Case Driven Approach*, ACM Press, Addison-Wesley, 1992.

Kidd, A. and Welbank, M., "Knowledge Acquisition" in *Expert Systems: State of the Art Report*, Pergammon Infotech, Maidenhead, Berkshire, England, 1984.

Koulopoulos, T.M. and Frappaolo, C., *Electronic Document Management Systems – A Portable Consultant*, McGraw-Hill, 1995.

Krichten, D.J., Wilson, K.D. and Tracy, K.D., "Expert System Guide Biological Phosphorous Removal", *Water Environment & Technology*, 3 (10), 60-64 (1991).

Larkin J. H. and Chabay R. W., eds., *Computer-Assisted Instruction and Intelligent Tutoring Systems: Shared Goals and Complementary Approaches*, Lawrence Erlbaum Associates, New Jersey, 1992.

Lewis & Zimmerman Associates Corporation, *WasteWater Training Program –WWTP™*, Lewis & Zimmerman Associates Corporation, 1996.

Martin, J. and Odell, J.J., *Object-Oriented Methods: A Foundation*, Prentice Hall, 1994.

Martin, J. and Odell, J.J., *Object-Oriented Methods: Pragmatic Considerations*, Prentice Hall, 1996.

Mowbray, J.S. and Malveau, R.C., *CORBA Design Patterns*, John Wiley & Sons, 1997.

Ozgur, N., Stenstorm, H. and Michael K., "Knowledge-based Expert System for Process Control of Nitrification in Activated Sludge Process", *J. of Environmental Engineering*, **120** (1) 1994, 87-107.

Patry, G. and Chapman, D. Eds., *Dynamic Modelling and Expert Systems in Wastewater Engineering*, Lewis 1989.

Patry, G. and Takács I, "GPS-X A Wastewater Treatment Plant Simulator", *Proceedings of the Conference on Mathematical Modeling*, February 2-4, Vienna, Austria, Vol. 3, 456-459 (1994).

Rao, B. R., *Object-Oriented Databases – Technology, Apllications, and Products*, McGraw-Hill, 1994.

Rogerson, D., *Inside COM*, Microsoft Press, 1997.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorenzen, W., *Object-Oriented Modelling and Design*, Prentice Hall, 1991.

Rumbaugh, J., *OMT Insights*, SIGS Books, 1996.

Sassen, J.M.A, Buiel, E.F.T., Hoegee, J.H., "Laboratory evaluation of a human operator support system", *Int. J. of Human-Computer Studies*, **40** (5), 895-931 (1994).

Shlaer S. and Mellor, S.J., *Object-Oriented Systems Analysis: Modeling the World in Data*, Yourdon Press, 1989.

Sigfried, S., *Understanding object-oriented software engineering*, IEEE Press, Los Alamitos, California, 1996.

Sommerville, I., *Software Engineering*, Fifth Edition, Addison-Wesley, 1995.

Sukaviriya, P.N., Isaacs, E. and Bharat, K., "Multimedia help: A prototype and an experiment", *Proceedings of ACM conference on human factors in computing systems (CHI 92)*, 433-444 (1992).

Vlissides, J.M., Coplien, J.O. and Kerth, N.L., *Pattern Languages of Program Design 2*, Addison-Wesley, 1996.

Water Environment Federation (WEF), *WEF Technical Library on CD-ROM*, WEF, 1996.

Water Polution Control Federation (WPCF), *Operation of Municipal Wastewater Treatment Plants: Manual of Practice*, Vols I & II, WPCF, 1990.

Wirfs-Brock, R., Wilkerson, B. and Wiener, L., *Designing Object-Oriented Software*, Prentice Hall, 1990.

[URL1] www.w3c.org

[URL2] www.rational.com/uml

Appendix A

THE UNIFIED MODELING LANGUAGE (UML)

In Chapter 3, several concepts of object oriented (OO) software analysis and design were introduced, and the unified modelling language (UML) was selected as the analysis and design language for this project. The reason for this selection is that software engineering, as any other field of engineering, needs a standard way for designers and implementers of software products to communicate clearly.

Until now such an effort seems out of reach due to the proliferation of OO analysis and design language and technology on the market. Companies wishing to use OO technology will either choose one of available offerings or create a new language to suit their own needs. The results are proprietary solutions and developers moving from one company to another have to relearn the same old concepts, but in new symbols, to be able to continue maintaining the software or to improve it. This is not something that the OO preachers are willing to admit publicly.

Then comes the UML. The UML is the result of the consolidating efforts of three leading OO methodologists: Grady Booch, James Rumbaugh and Ivar Jacobson. Suddenly a *defacto* standard emerges, which builds momentum and is being ratified by the Object Management Group (OMG) as a *dejure* standard.

This means that analyses and designs developed using the UML will be useable to a greater audience and are accessible to a variety of tools that are being developed.

The UML at the core consists of three components: a system of largely graphical symbols and notational constructs, a semantic specification on the meaning of the symbols and a suggested methodology. The graphical symbols and notational constructs provide standard ways to represent many well-known OO concepts as well as provisions for proprietary expansion (*via* notational constructs). The semantic specification is necessary for appropriate use and correct interpretation of the symbols. However, the suggested methodology is optional since it is a path of steps to follow on how to best use the UML that may or may not be useful in every case. Therefore, to benefit from the UML it is sufficient to use the symbols and notational constructs along with an understanding of the semantic specification. The user can then apply his/her own favourite analysis and design method. Such use of the UML is exemplified in this project, where a combination of design patterns and traditional methods were used in the analysis and design, while the results were expressed in the UML.

The UML is large and rich, and an in-depth treatment on its various aspects can be found at the official Rational web site (URL1) or in Fowler and Scott (1997). For the remaining of this appendix, a description of the elements of

the UML used in this project will be made available so that the interest reader can better understand the ideas presented.

Use-case[1]

For a long time, people using either the traditional or OO approach to system analysis have been using scenarios to help them understand the requirements. However, these activities were largely informal and undocumented. Jacobson (1994) formalises the activity of creating scenarios into a documented step in the software engineering process. He called the scenario in his term: 'use case'. The object community adopted 'use case' to such an extent that it has become an essential part of the analysis, and now it has made its way to the UML.

A use-case is essentially a typical interaction between a user and a system. The system can be a computer or a process. Examples of use-cases include "create a new water record registration" and "edit a measurement". Use-case has the following properties (Fowler and Scott, 1997):

- A use-case captures some user-visible function.
- A use-case may be small or large in scope.
- A use-case achieves a discrete goal for the user.

[1] The official syntax is use case without the hyphen. However, I have written the word with a hyphen to lessen its bad English construct. The meaning of either version remains the same.

Use-case properties as enumerated above are most applicable to situations where people are involved. However, the benefit of use-case does not stop there. Use-case can also be employed to describe interactions between system components. In short, use-case is a formal tool to capture and describe high level interactions between participants of a system. These participants are called *actors* in a use-case.

Jacobson (1994) introduced the use-case diagram as a visualisation tool for use-case. The elements used in this project are illustrated in Error! Reference source not found.. In the figure, there are three different kinds of actors: human, software and system. There are communication directions: uni-direction shown as a single-headed arrow and bi-direction shown as the double-headed arrow. A use-case can also be inherited from a more general one to express a situation where a scenario may be the derivation of something more basic. For example, the "create an observation" and "create a measurement" can both be derived from a generic "create an object" use-case. In general, the use-case is read as "actor + communication path + other actor (optional) + the content of the use-case". For example, in **Figure A – 1** the human actor communicates with the GUI system that communicates with the search server to activate the "Make available specific search server" use-case.

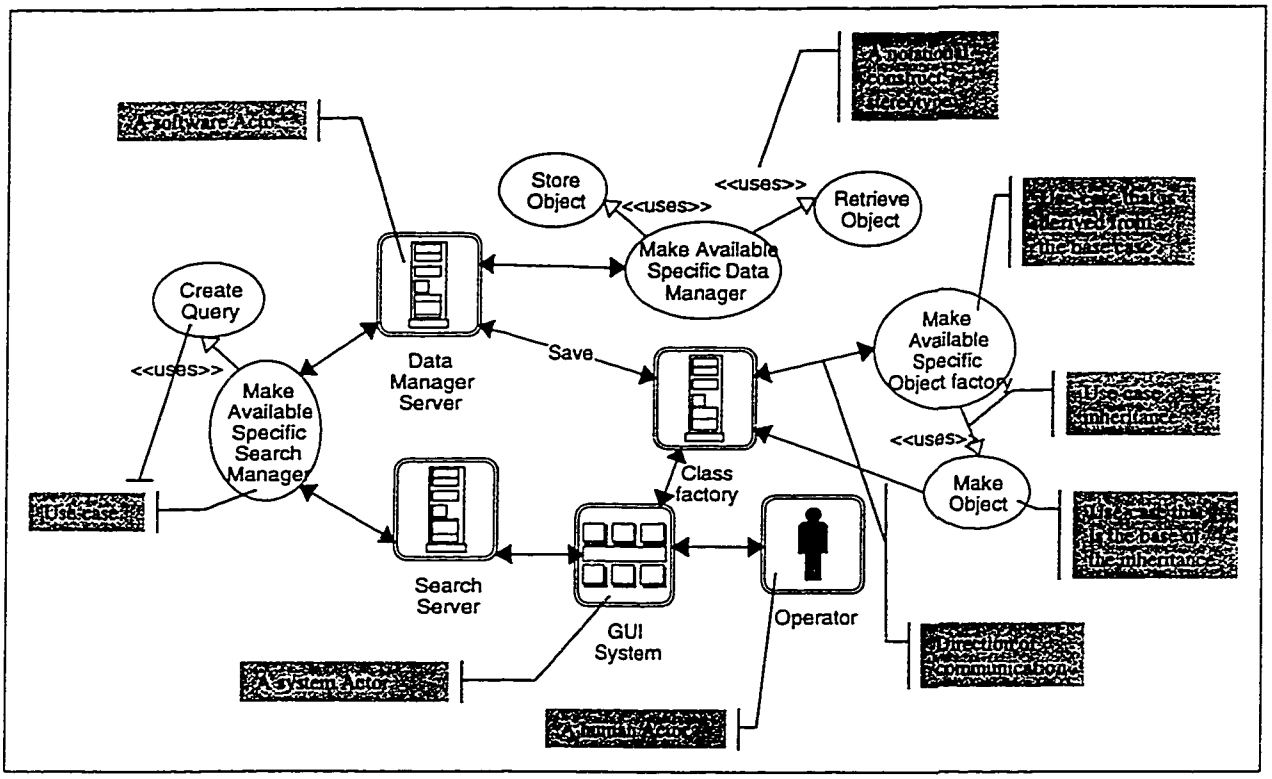


Figure A - 1 Elements of use-case diagram employed

There are notational constructs "<<uses>>" that appear in **Error! Reference source not found.**. They are called stereotypes and are examples of a UML facility for extending the meaning of the relationship between entities in use-case and elsewhere. There are other constructs for use-case that are not discussed because they are not used in this project. They can be studied elsewhere (Rational, URL2; Fowler and Scott, 1997).

Class diagram

Class diagrams are central to OO methods. A class diagram describes the types of objects in the system and the various kind of static relationships that

exist among them (Fowler and Scott, 1997). There are two principal kinds of static relationships:

- Associations (an operator collects one or more water samples).
- Subtypes (an operator is a kind of user of the system).

Class diagram can also show the attributes and the operations of a class and the constraints that apply to the way class of objects are connected. **Figure A – 2** shows a typical class diagram.

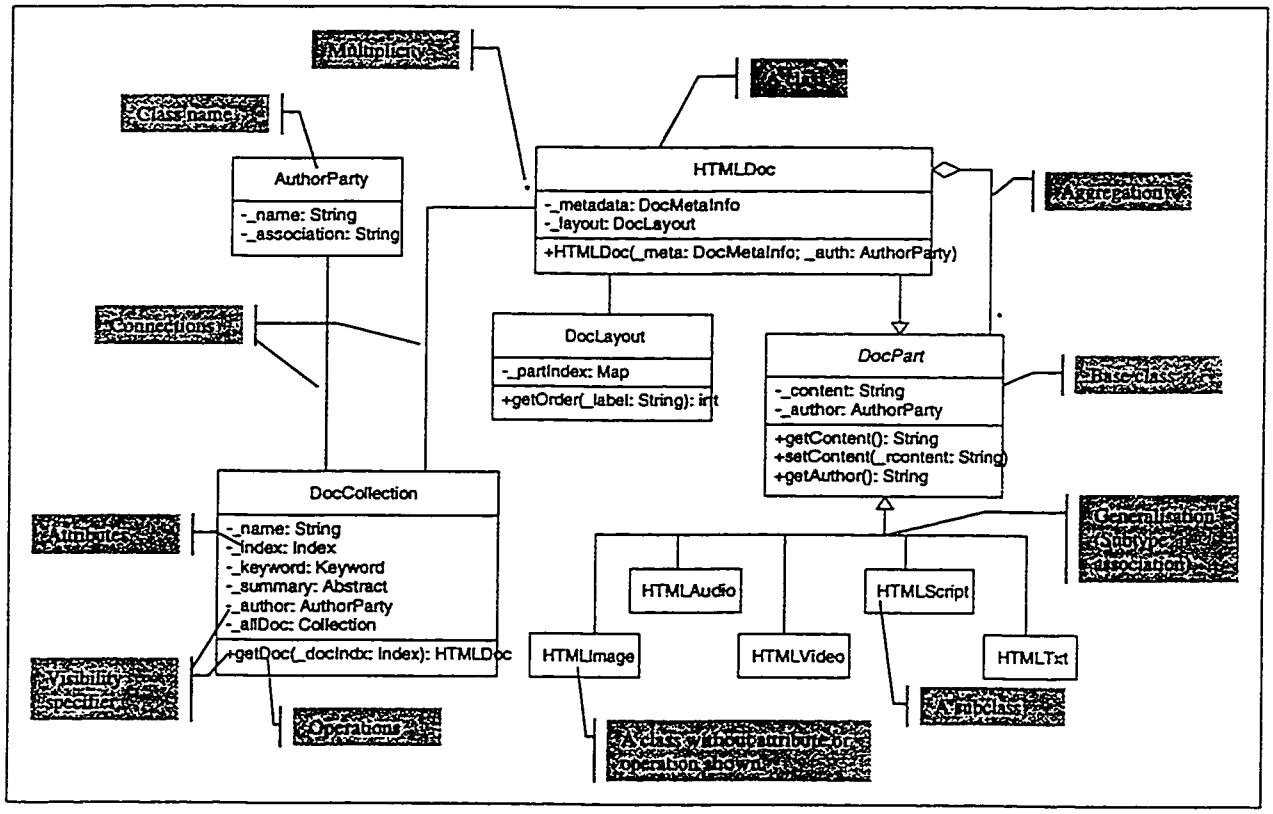


Figure A - 2 A class diagram

There are three perspectives to class diagram that must be clearly distinguished when creating or reading a class diagram: conceptual, specification and implementation (Cook and Daniel, 1994; Fowler and Scott, 1997). With the

conceptual perspective, a class is drawn without any details about its attributes or operations (see a sample within **Figure A - 2**). The class diagram is concentrated on the domain under study. What to note on a conceptual diagram (or model) are the logical relationships that exist among classes of the domain. The classes are presented independently of any implementation details such as the choice of software or language used.

With the specification perspective, the focus is on the interfaces of software components, not the implementation. Types are examined instead of classes. A type represents an interface that may have many implementations, different due to implementation environment, performance characteristics or vendor. Object-oriented development puts great emphasis on the interface and the implementation, but this is overlooked in practice because the notion of a class in an OO language combines both interface and implementation. On the other hand, the implementation perspective is about classes, with their inner construct exposed and ready for the software construction phase. The latter is usually found in software maintenance documents, whereas class diagrams with a specification perspective are mostly found at the design phase.

With the perspective clearly distinguished, the elements shown in **Figure A - 2** can be described. These are *class, attributes operations, visibility, associations, multiplicity (cardinality), generalisation* and *aggregation.*

Class

A class is represented in the UML as an enclosed rectangular that has the format shown in **Figure A – 3**. A class can be shown with a name only when used conceptually. A class is associated with other classes. These associations represent the relationships between classes. Furthermore, a class represents a group of similar objects. These objects are instances of the class. To describe the common characteristics and behaviours of objects, a class has attributes and operations.

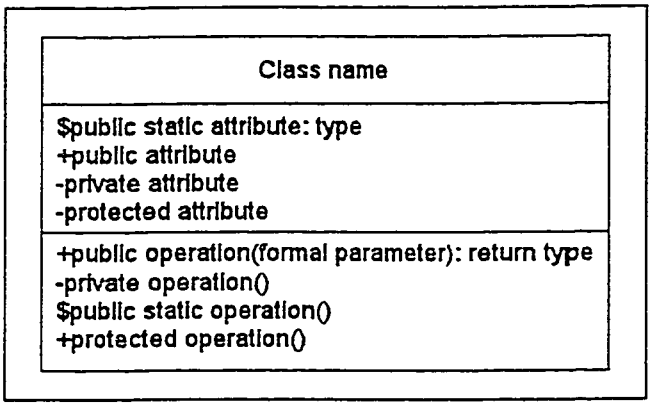


Figure A - 3 A UML class

Attributes

At the conceptual level, the presence of an attribute signifies simply that that class has it. At the specification level, this attribute indicates that the class can tell the outside about its nature and has someway to adjust its value. And at the implementation level the same attribute reduces to a variable name or a field with language specific constructs. Of course, attributes can be hidden to the outside altogether by declaring them **private**. Their natures can be visible by

declaring them **public**, or they can be available to a few selected classes by declaring them **protected**. Private, public and protected denote visibility, which is the same for operations (describe in next section). UML 1.0 puts a negative (-) sign to denote a private declaration and a positive (+) sign for both public and protected declaration (**Figure A - 3**). There is also a **static** visibility (a dollar sign ($) in UML 1.0) to denote an attribute common to all instances of the class. Depending on the detail in the diagram, a type of the attribute in addition to the name can also be specified. The UML 1.0 syntax for attribute is *visibility name: type = default value.*

Operations

In OO parlance a class has states (attributes) and behaviours. **Operations** are the processes that a class knows how to carry out. They are also called methods or member functions of a class, but these have language dependent connotations ("method" is used in SmallTalk and Java and "member function" is used in C++). When an object is said to send a message to another object, the sender object is actually invoking a method on the receiver object. Operations are rarely shown with a conceptual perspective. If they do, they represent the responsibilities that a class has. They are found mostly in the specification and implementation perspectives. In specification model, operations correspond to public method on a type (*i.e.*, interface of class). At the implementation level, private and protected operations are also shown.

The full UML 1.0 syntax for operation is:

visibility name (parameter-list): return type expression {property string}

Where

- visibility is the positive sign (+) for public, the negative sign (-) for private, the pound sign (#) for protected, or the dollar sign ($) for static.
- name is a string.
- parameter-list contains (optional) parameters whose syntax is the same as that for attributes.
- Return-type-expression is optional, language-dependent specification.
- Property-string (optional) indicates property values that apply to the given operation.

An example of an operation may be *+getdoc(_docIndx: Index): HTMLDoc* (**Figure A - 2**).

Associations

Associations represent relationships that exist between instances of a class (a document collection has documents, a document is composed of several parts). From a conceptual perspective, associations represent conceptual relationship between classes. Within the specification perspective, associations represent responsibilities. Whereas the same associations may be represented by a bi-directional pointer or reference, or a method call in the implementation perspective.

Each association has two **roles;** each role is a direction on the association. Each association can have a label. It can also be shown with **navigability;** that is, the association can be uni-directional or bi-directional. Navigability denotes whether a class can tell about (or give access to) its partner in a relationship. Navigability is shown with arrows, a single-headed arrow for uni-direction and a double-headed arrow for bi-direction. In the UML 1.0, association without arrow head(s) is assumed to have navigability unknown or bi-directional. A role also has **multiplicity,** which is an indication of the number of objects participating in a given relationship. In **Figure A - 2,** the '*' between the DocCollection class and the HTMLDoc class means that one instance of DocCollection knows about many instances of HTMLDoc. Possible multiplicity notation in UML 1.0 is shown in **Figure A - 4.** Multiplicity is put close to the class that it modifies. If multiplicity is not shown, it is assumed to be one by convention.

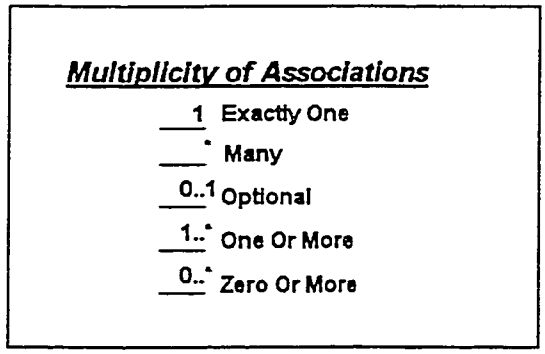


Figure A - 4 UML notation for multiplicity of associations

Generalisation and aggregation

Generalisation and **aggregation** are special kinds of association. Generalisation is used to represent the concept of similar characteristics and

behaviours among classes. It has a directional meaning where something is more general than something else. In contrast, the term specialisation is used for the reverse direction to denote something which is more specialised than something else. A mammal class is a generalisation of a large group of animals that share certain characteristics and behaviour. However, a lion is a specialised 'version' of the feline family (**Figure A – 5**). Within a specification perspective, generalisation means that

all the interfaces of the super type can be found in the interface of the sub-type. Generalisation at the implementation level means inheritance at the language level (a specialisation, a class inherits from its super class and thus is more specialised).

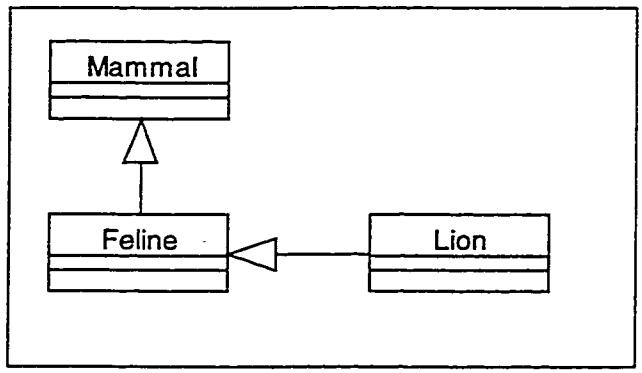


Figure A - 5 A generalization association

Aggregation denotes an inclusion relationship or a part-of relationship. The UML also makes available **composition**, which has a stronger meaning than aggregation. Plain aggregation is used to describe a relationship such as a country has citizens. However, composition is used to express a strong cohesion such as an engine has parts. The engine can not function efficiently without one

of its parts. Therefore, if a class is composed of parts, deleting that class also includes deleting all of its parts. In contrast, an aggregation has a more relaxed meaning. The HTMLDoc class in **Figure A - 2** has an aggregation relationship with its DocParts (**Figure A – 6**), which means that an HTMLDoc instance can be deleted without affecting the DocParts. This usage is necessary because a DocPart can be used in another HTMLDoc instance.

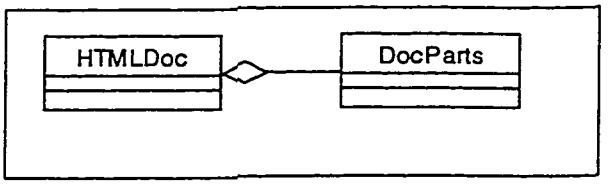


Figure A - 6 An aggregation association

Proper usage of aggregation requires discipline because an aggregation relationship can mean several things at the implementation level. Coad and Mayfield (1996) recommended the use of composition to reduce coupling due to generalisation. However, their implementation examples showed what others would consider aggregation (Booch, 1994; Rumbaugh *et al.*, 1991). Therefore, the meaning of aggregation and composition must be clearly established for team usage (Fowler and Scott, 1997).

Package diagram

Package diagram such as the one shown in **Figure A - 7** is a UML mechanism for organising classes into related groups. The purpose of a package diagram is to show how the overall software system is composed, without going

to the details about how the classes are related to each other. This is an effective tool that a designer can use to show the dependency of various packages in a system. This helps to achieve what Booch (1996) calls architecture evaluation. This evaluation helps identify key structures for system construction and to ensure that future maintenance or overhaul of the system can proceed without accidentally affecting important sub-systems.

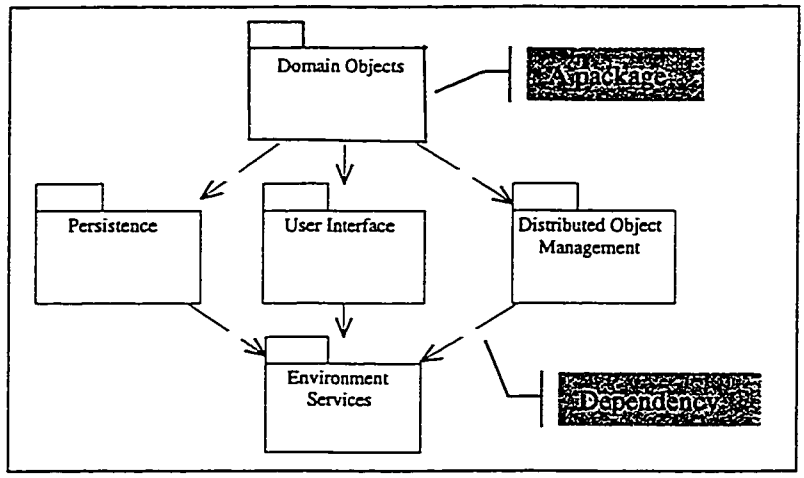


Figure A - 7 A package diagram

Further study

This concludes the brief tutorial into the UML 1.0 elements that are of significance to this thesis. The UML is a rich modelling language that has other elements such as sequence diagram, interaction diagram, activity diagram and deployment diagram. These additional elements found their use mostly in detailed design and implementation documents that accompany the software lifecycle. However, their inclusion would result in too many details about the implementation process, which could change as technology evolves and

prototypes are developed and thrown away. Interested readers can always inquire more about these and other UML elements from the references provided.

APPENDIX B

THE MICROSOFT COMPONENT OBJECT MODEL (COM)

In **Chapter 5**, component oriented software development was chosen as the implementation style for the prototype system. The Component Object Model (COM) standard and COM Automation from Microsoft were used. COM Automation technology is an implementation of the COM standard that provides software "parts" to be assembled into full application, using tools such as Visual Basic and VBScript. This appendix offers a brief tutorial into aspects of the COM standard and the COM Automation component developmental process. The goal is to provide the interested reader with relevant background information to better understand the thesis and with starting points into the technologies for further personal interest. The COM standard and its associated technologies are complex and are very important to Microsoft's future, as such its in-depth treatment can be better found in Rogerson (1997).

What is the Component Object Model (COM)?

A standard must exist in order to build things using parts. This standard would specify rules and requirements such that individual parts made by many manufacturers could work together. COM is such a standard produced by Microsoft in order to build parts for Windows software. COM is a specification of how the interface between software components must be implemented such

that they can work together with other software components that require the same interface.

Software components that are developed using the COM standard are called COM components. COM components are language independent. Therefore, any language can be used to develop them. The advantage of language independence is that the strengths of various languages can be put to good use. One such example is the prototype of this project. Java was used to implement the COM component that worked with an object oriented database engine (*i.e.,* PSE) because it has strengths such as automatic memory management, is simpler compared to C++ and has a better object model than Visual Basic. Java codes can also be tailored so that reuse on another platform is possible. On the other hand, Visual Basic was used to develop the user interface because it is most suitable for such a development on the Windows platform in terms of development time saving. Using Visual Basic also means that existing components will be available for other tasks such as chart display or relational database integration.

Other benefits for COM components include a binary standard, easy upgrade and transparent network support. A binary standard allows components to be shipped in binary form, developers of components can be confident that their parts will work without providing their clients source codes. COM components are easy to maintain because old interfaces are warranted to

work in a newer version of the component. Developers can add features to the component without requiring the client software to be recompiled. COM components can be used transparently over the network as if they were local. This is possible because the COM support library ensures that COM components behave in the same manner to the client, whether they reside on the same machine or are located on another machine across the network.

So what exactly is the COM standard composed of? In a nutshell, COM specifies that support system for component development must provide tools for specifying the interfaces. COM specifies a minimal set of interfaces that all interfaces would be derived from (*e.g.*, Interface Unknown (IUnknown) and Interface Dispatch (IDispatch)). For example, all interfaces of COM components are derived from IUnknown. This includes an interface query mechanism (QueryInterface) so that the client can find out about available interfaces and how to use them and a reference count mechanism to help with memory management. The COM specification also includes a mechanism to create instance of components (ClassFactory), a mechanism to register the components, a standard way to uniquely identify component (Globally Unique Identifier (GUID)), a standard way to communicate error and internal state (HRESULT) and a threading model for multi-thread support.

There are tools such as the Microsoft Interface Definition Language (MIDL) and the Object Definition Language (ODL) to specify COM component

interfaces, the registry to register components and others that are forthcoming from Microsoft. At the time of this writing, Microsoft announced the accomplishment of COM+ and the Distributed Network Architecture (DNA) that include a better COM specification for the internet and DNA tools such as Active Directory and Transaction Server to provide the backbone of component software development for the Internet and beyond. It is beyond the scope of this brief tutorial to go into detail about each of the above mentioned elements of COM. A better reference for COM in general is Rogerson (1997) and Brockschmidt (1996) for development of technologies leading to COM and the technology of COM itself, but the best source will be the Microsoft web site for the latest information on COM, COM+ and DNA. In the remaining sections, the COM Automation developmental process with Java will be explained. This was the process that was used in the prototype implementation of this project.

COM Automation

The term *Automation* is a legacy of the previous COM evolution namely OLE 2.0[1]. While a minimal COM component must implement the IUnknown interface, a COM Automation component must also implement the IDispatch interface. A COM interface is essentially a pointer to an array of function pointers at the binary level. To use a COM interface a client must know the

[1] OLE used to stand for Object Linking and Embedding for version 1.0 and reflects the original intention of the technology. However, as OLE progresses and becomes much more Microsoft stops giving meaning to the acronym and refer to it as simple OLE, until it becomes COM.

layout of this array in advance either statically (static binding) or dynamically (dynamic binding). That is, the client must know which functions occur in which order and what parameter and return types they have. Tools that assemble components to form application (*e.g.,* Visual Basic) must have facility to accommodate this knowledge.

Using components in statistically binding languages like C/C++, this knowledge can be accommodated readily *via* IUnknown. However, since Visual Basic and Java are interpretive languages, components written in either language[2] need a different mechanism to permit dynamic interface usage. Such facility is the IDispatch interface. This interface allows the client to access the functions of the COM component interfaces indirectly. The functions of the COM component interfaces are accessible through the standard functions of IDispatch[3].

There are several ways to implement IDispatch, and Microsoft differentiates two by name: *dispatch interface* or *dispinterface,* and *dual interface* or just *dual.* The dispinterface implementation only supports indirect interfaces referencing. On the other hand, the dual implementation not only has the same supports as dispinterface, it also supports direct interfaces referencing by allowing a direct access to the interfaces that inherit from IUnknown. Indirect

---

[2] Version 5 of Visual Basic provides this ability. Prior to version 5, Visual Basic can only assemble and run components, not writing them from scratch.

[3] The details of these standard functions are unfortunately a bit lengthy to be included in this appendix. For the time being it is sufficient to know what they are for.

access to interfaces *via* dispinterface is more expensive and slower than direct access *via* IUnknown, but it is necessary for interpretive environments like Visual Basic and Java. However, developers will want to maximize the use of their components; therefore, the dual implementation is the most popular way to implement IDispatch.

A Visual Basic, C++ or Java program can control a component *via* a dispinterface without any type information about the dispinterface or its methods. This is possible, but it entails significant overhead work from the integrator of the components in implementing run-time type checking and conversion. To aid the component integrator, Microsoft provides a solution in COM called the *type libraries*. Type libraries provide type information about components, interfaces, methods, properties, argument and structures. A type library is a compiled version of the Interface Definition Language (IDL) file, is in binary format and can be accessed programmatically. Type libraries also contain help strings for all the properties and methods, so that a browser inside Visual Basic (or any compliant browser) can display help on any property or method to user of the component. Microsoft provides tools to handle type library creation and reading in the Automation library.

Microsoft support for Automation using Java

Version 2.0 of the Java Virtual Machine (JVM) from Microsoft provides a feature called "Automatic IDispatch", which allows any Java class to present an

IDispatch interface. This is done simply by registering the class using the *Javareg* tool, which is provided with the software development kit (SDK), and the Java object is registered as a COM object. Any Automation client can then create a Java class registered this way. This is a nearly effortless way to create Automation objects from Java classes, but it has several limitations. First, only dispinterface is supported and the dual interface is not possible this way. Second, Automation objects generated this way do not work smoothly with a client such as Visual Basic because certain standard conventions can not be enforced. To have the benefits of a dual interface and ensure that Visual Basic will have no problems working with Java Automation object, the developmental effort must begin with creating an IDL file using the Microsoft IDL syntax. The steps of the process are shown in **Figure B - 1**.

After the IDL file is created by the developer, the MIDL compiler is used to generate the type libraries files. The type libraries are packaged into a file with the extension '.tlb'. Following this step, the JActiveX tool is used to generate the skeleton Java files with COM directives embedded. At this stage, the developer has to implement the public Java class methods that make up the exposed interface. It is at this stage that library classes are combined to create the Automation component. Library classes are such as those provided by the core Java packages, and specialised classes are such as those provided by the PSE object-oriented persistent engine. The resulting Java files are compiled by the Microsoft Java compiler to produce the usual Java byte-code classes.

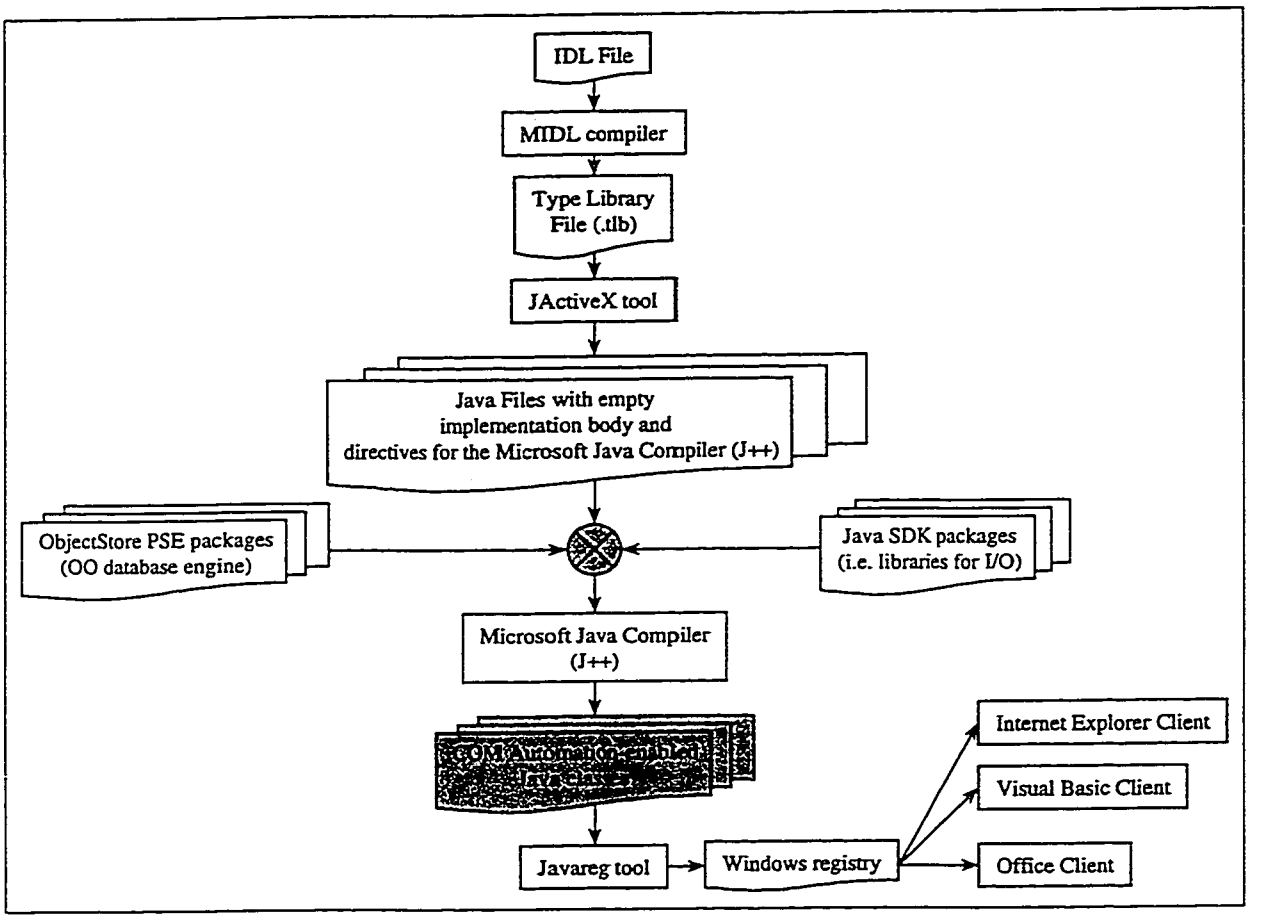
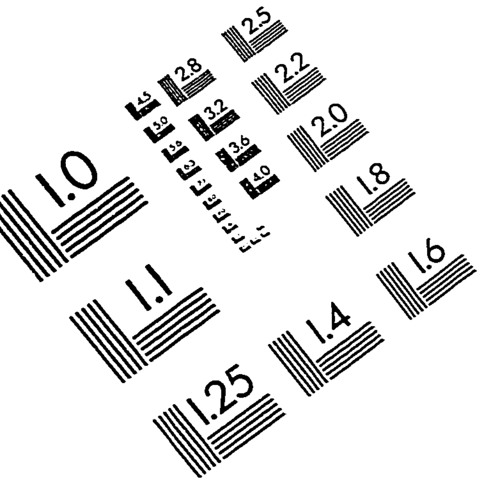
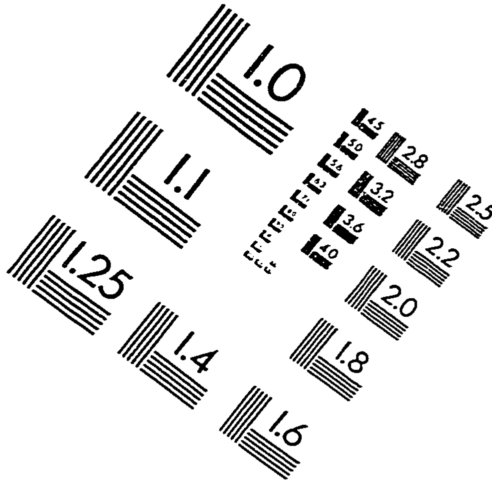


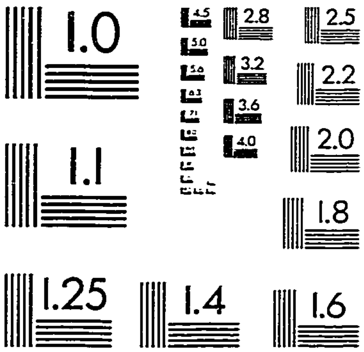
Figure B - 1 The COM Automation development process with Java

The classes are COM aware after this stage and needed to be registered with the Windows registry before a client such as Visual Basic can see and use them in application development. However, access to these component functions is not limited to Visual Basic. As **Figure B – 1** shows, any COM Automation client (*e.g.*, Internet Explorer, Excel and Word) can now use the newly created component. This is one of the major advantages of using COM.

Index

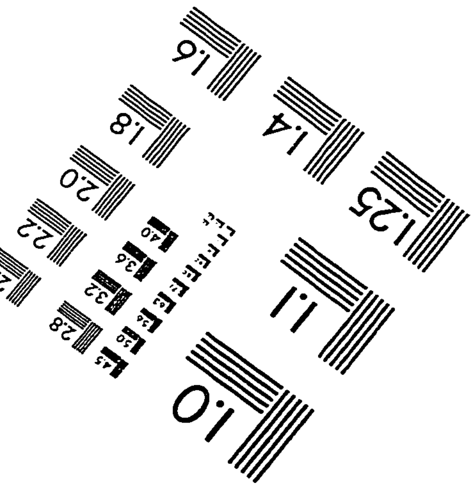
IMAGE EVALUATION TEST TARGET (QA-3)



150mm

6″



APPLIED IMAGE . Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989