THE DEVELOPMENT AND IMPLEMENTATION OF AN *INTELLIGENT, SEMANTIC MACHINE CONTROL* SYSTEM WITH SPECIFIC REFERENCE TO INFORMATION SYSTEM ARCHITECTURE DESIGN

BY

JIANG WU

A DISSERTATION PRESENTED TO THE HIGHER DEGREES COMMITTEE OF THE CAPE PENINSULA UNIVERSITY OF TECHNOLOGY IN FULFILLMENT OF THE REQUIKEMENTS FOR THE MASTER OF TECHNOLOGY: INFORMATION TECHNOLOGY DEGREE.

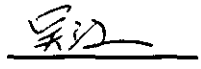
CAPE PENINSULA UNIVERSITY OF TECHNOLOGY

2005

DECLARATION

The contents of this dissertation represent my own work, and the opinions contained herein are my own and not necessarily those of the university. All references have been accurately reported.

Name: **Jiang Wu**

Signature: 

Date: **March 2006**

Acknowledgements

I wish to express my sincere acknowledgement and appreciation to the following people:

Mr. Bennett Alexander, my supervisor, for all his help, enthusiasm and guidance and for the encouragement to pursue this project.

I would also like to thank my colleagues at the department, in particular JiaChun Wu for his cooperation and fruitful discussions.

All other Cape Peninsula University of technology, for their friendship, support and motivation.

My parents, for encouragement, motivation and support.

Feng, my wife, for endless patience, encouragement and love.

致谢

我十分感谢我的导师，Alexander 先生，感谢他给予我大力的支持和鼓励，以及对论文提出的建设性批评和意见。我要感谢一直陪伴在我身边的我的妻子给予我的学习和生活上的帮助。我还要感谢所有在南非帮助过我的人们。最后，我要感谢在中国的父亲、母亲以及亲人、朋友，感谢他们的支持和鼓励。

Abstract

This thesis explores the design and implementation of an intelligent semantic machine control system with specific reference to information system architecture design. The term "intelligent" refers to machines that can execute some level of decision taking in context. The term "semantic" refers to a structured language that allows user and machine to communicate.

This study will explore all the key concepts about an intelligent semantic machine control system with information system architecture. The key concepts to be investigated will include Intelligent Control, Semantics and information system architecture. The primary purpose of this study is to develop a methodology for designing information system architecture.

The emerging discipline of "Usability Engineering" is at the core of information system architecture aspects of this project. The Usability Engineering approach to the design of complex machines focuses on developing machines that are efficient and error-free. Usability Engineering provides a methodological framework for the optimum design of information system architecture by recognising - user needs, design restrictions, and other environmental constraints. The Usability Engineering also provides guidelines for integration with Object Oriented Methodology (OOM) and Unified Modelling Language (UML). The integration is based on linking OOM models and UML with Usability Engineering tasks. OOM is a new technology based on objects and classes. By providing first class support for the objects and classes of objects of an application domain, the object-oriented paradigm precepts offer better modelling and implementation of systems. The UML is an open method used to specify, visualize, construct, and document the artifacts of an object-oriented information system under development.

This study illustrates the design and implementation of information system architecture of an intelligent exercise machine as a specific practical Information Technology (IT) application. We will follow the integration of the Usability Engineering and OOM to develop the specific application. The intelligent control system will automatically respond and execute a task or a function of the machines immediately in terms of decision taking of machines. The important aspect of the information system is to record all users' data for customizing their future plans and retrieving the data. Information system architecture provides a communication between an exerciser and a coach in terms of exerciser needs.

Abbreviations

BAD: Business Architecture Definition

DAQ: Data Acquisition

DEMS: Data Exercise Management System

ELF: Elementary Loop of Functioning

IA: Information Architecture

IT: Information Technology

MRR: Material Removal Rate

OO: Object-oriented

OOM: Object Oriented Methodology

TAD: Technical Architecture Definition

UML: Unified Modeling Language

Table of Contents

List of Figures

CHAPTER 1 INTRODUCTION

1.1 Overview

The fitness industry is lagging behind on incorporating Information Technology into its operations. As we investigated, exercisers have to adjust the weight physically, count the exercising times by themselves, remember all the exercise items, and bear no feedback, when they are doing exercises with the regular exercise instruments. The exercisers hope the fitness industry could supply a multifunctional exercise machine to help them to concentrate on the exercise only. From this purpose, this project aims to develop information system architecture for a novel intelligent exercise machine control system. The exercise machine control system will have an advanced diagnostic capacity for monitoring users'body status. The Information system architecture of the exercise machine will have well-defined features and functionalities in order to maximally satisfy all different users' needs and optimize exercise effectiveness for exercisers.

Meystel A. and Messina E. (2000) indicated: "Intelligent Control is a commonly used term, applied to an amorphous set of tools, techniques, and approaches to implementing control systems that have capabilities beyond those attainable through classical control. There is a well-developed and deep body of work entailing a theory of control. This is not the case for intelligent control, which is more of an ad hoc approach to constructing systems that generally contain some internal kernel of the more classical control, for which there is a theory."

The dissertation covers the introduction of the study: the reviews of intelligent machine, semantic machine, machine control and information system architecture; the discussion of methodology of Object-Oriented design and Unified Modelling

Language; the analysis of implementation of the information system architecture; the techniques of using .NET and LabView; and the implementation of information system.

The research represents the consideration of the problem, namely that the fitness industry is lagging behind on incorporating Information Technology into its operations. An experimental research method was used, with the aim of solving the above problem. In other words, an information system architecture had to be designed and implemented to function within the Data Exercise Management System.

This information system architecture can be used for communicated exercise and fitness data, via an Intelligence Trading House, to Doctors and Coachers. It provides facilities to novelty offering useful empirical information on exercise status to interested third parties.

1.2 Background

The increasingly sedentary lifestyle of the past 30 years has contributed to declining health. Consequently exercise is vital for good health maintenance. Optimally managed exercise with accurate information feedback is needed. Compared to other industries, the fitness industry is lagging behind on incorporating Information Technology into its operations. For example, the International Health Racquet and Sports-club Association (hereafter referred to as the IHRSA) in March 2005 convened only their first technology summit entitled "Generating Revenue & Enhancing Profitability Through Technology" (keynote address given by Microsoft Business Solutions). This indicates that it is necessary and current to offer IT in fitness management.

This project developed a novel dynamic resistance exercise machine with onboard

data management. This new technology optimized exercise effectiveness for users. Our exercisers had advanced diagnostic capacity, and captured user data that can be manipulated into meaningful information for use by Doctors and Coachers.

This project established a full integrated technology that combined: dynamic resistance exercising with the electronic capturing / managing of data drawn from the machine user. This required a paradigm shift from single function exercise equipment to intelligent exercise equipment integrated within an IT based information network.

In this project, Information System's driven and pneumatically actuated exercise equipment was a forward-facing unit. Onboard was the Data Exercise Management System known as DEMS. The DEMS communicated exercise and fitness data, via an Intelligence Trading House, to Doctors and Coachers. This project novelty lies in the exercises configured for individuals, which allows for seamless exercise routine, without the need for physically adjusting 'weights'. DEMS is an additional novelty offering useful empirical information on exercise status to interested third parties.

1.2.1 Intelligent control system

The ever increasing technological demands of today call for very complex systems, which in turn require highly sophisticated controllers to ensure that high performance can be achieved and maintained under adverse conditions. There are needs in the control of these complex systems which cannot be met by conventional approaches to control. For instance, there is a significant interest in enhancing current avionic systems so that they can reconfigure the aircraft controls to maintain adequate levels of performance even if there are complete failures in one or more of the actuators or sensors. In a similar manner, there is a significant need to achieve higher degrees of autonomous operation for robotic systems, spacecraft, manufacturing systems, automotive systems, underwater and land vehicles, and others. To achieve such highly autonomous behaviour for complex systems one can enhance today's control methods

using intelligent control systems and techniques (Polycarpou M. n.d.).

1.2.2 Machine control system

For a great demand for servo control systems in the field of manufacturing and industrial processing control, the commonly used systems are electric servo systems, hydraulic servo systems and pneumatic servo systems. Generally electric and hydraulic servo systems are used to meet the need of higher performance with higher costs. The pneumatic servo systems are normally used to meet the need of lower performance because they have the advantages of lower cost, of long working life, of working overload, of anti-explosion and without the need of maintenance. But on the other hand, the control accuracy is affected badly by its nonlinear characteristics. Fortunately the application of a fuzzy procedure in conventional regulators is a very simple way to improve the performance of a conventional control strategy. If some robust and adaptive controllers can be found to compensate the influence of the nonlinear characteristics, the control accuracy will be enhanced and the pneumatic servo systems will have better prospect of application in the future (Sloman 1994).

1.2.3 Information system architecture

At its most basic, Information Architecture is a field and approach to designing clear, understandable communications by giving care to structure, context, and presentation of data and information (Poggio n.d.). It is the term used to describe the structure of a system, i.e. the way information is grouped, the navigation methods and terminology used within the system (Barker n.d.). It is the process of developing the information structure of the system, which takes place during the Define phase, with the aim of creating an effective system and a good user experience. Outputs of this process normally include the system plan, wire frames and parts of the functional specification (Public Life n.d.). It is the art and science of structuring knowledge, and defining user interactions (Brainboost n.d.).

The most common problem with information architectures is that they simply mimic a company's organizational structure. Although this can often appear logical and an easy solution for those involved in defining the architecture, people using systems (even intranets) often don't know or think in terms of organizational structure when trying to find information (Barker n.d.).

1.3 Statement of the Problem

This study is a development of a practical Information Technology architecture, involving the design and implementation of an intelligent semantic machine control system.

People do exercise in general gym, but they can't save any information and reuse it efficiently. So a requirement for optimally managed exercise with accurate information feedback is needed.

The purpose of this project is to develop an information system to support the fitness industry.

The above mentioned problem will be subdivided into the following sub-problems:

1. monitor exercise real time and remotely
2. make an exercise plan for specific exerciser
3. capture data from database
4. set default exercise mode and intensity for a new exerciser
5. set exercise mode and intensity to follow the plan that the coacher has made
6. set exercise mode and intensity by existent data of exerciser
7. set exercise mode and intensity manually

For this purpose, we need to investigate all important concepts in this work, such as intelligent control system, Semantic in an intelligent control system, Machine control system, Information system architecture at first.

Then we work out a suitable methodology and framework for the design of the Information system architecture.

Last but not the least, we develop an Information system architecture of an intelligent, semantic machine control system as a practical project to against the methodology and framework we used.

1.4 Causal Factors

1. The exerciser needs to exercise special muscles with dynamic resistance exercising;
2. The exerciser needs to meet his fitness objectives more seamlessly;
3. Medical specialists and fitness consultants need via technology inputs to define the dynamics of exercise;
4. The data statistics of machine users need to be remotely stored for manipulation by various third parties.

1.5 The research logic model

The logic model is an effective tool that helps to lay out the groundwork for the evaluation of the outcome of an intervention, which is an important factor in determining future success in development and action research (Bytheway A. 2002).

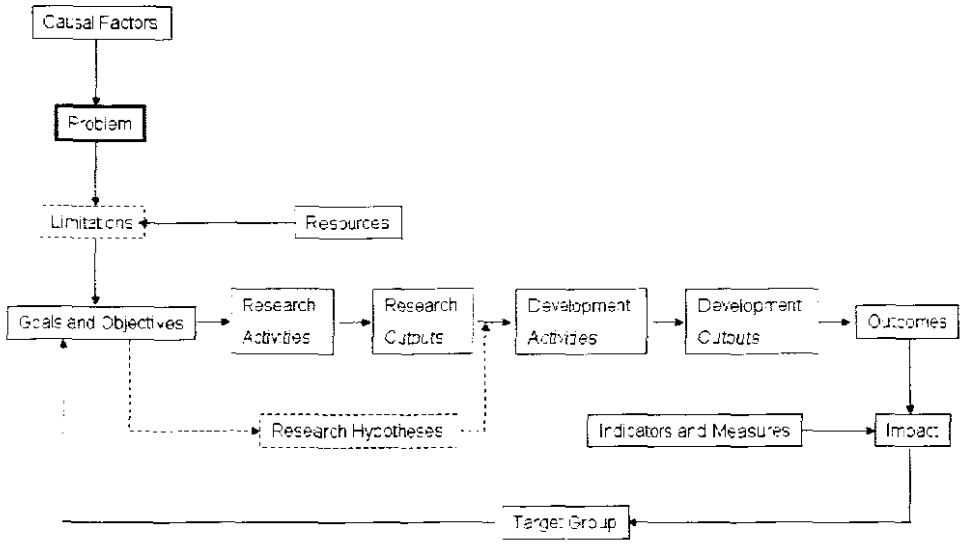


Fig1.1 Logic model (Bytheway A. 2002)

Fig1.1 shows the main features of the simple logic model. There are six components of particular interest that are largely self-evident. The detailed components of the model are discussed in the followed section (Bytheway A. 2002).

1. **Goals and objectives** are aimed or desired results what one hopes to achieve.
2. **Outputs** are what the activities produce.
3. **Outcomes** are what will be expected to happen with the available outputs.
4. **Activities** are what are to be done in order to achieve the goals and objectives
5. **The target group** is the community or population of people (or organisations) that **are the** intended beneficiaries.
6. **Resources** are what are needed to carry out the activities.

1.5.1 The significance of the research (objectives, goals, outputs, outcomes)

The following objectives were determined for the research:

1. Design a module for controlling a Pneumatic Actuator;
2. Design a module for capture data from the exercise machine;
3. The exercise machine supplies a series of exercise modes;

4. The exercise machine supplies a series of intensities for different exercise modes;
5. The exercise machine supplies a function for choosing an exercise mode and intensity for the exerciser;
6. The exercise machine supplies a function for Fail-Safe operation;
7. The exerciser stores his data of exercise in this database system;
8. Doctor and coacher capture data from database to analysis for the exerciser;
9. Coacher monitors and supervises exercise real time;
10. Coacher makes an exercise plan for next time;
11. The exercise machine sets default exercise mode and intensity for a new exerciser;
12. The exercise machine sets exercise mode and intensity to follow the plan that the coacher was made;
13. The exercise machine sets exercise mode and intensity by existent data of exerciser;
14. The exercise machine sets exercise mode and intensity manually.

The construction of the Intelligent Exercise Machine was as followed:

The flow chart Fig1.2, includes normal functions for the user at an exerciser level. The exercise machine supplied two ways to choose the mode and intensity for exerciser, one way was manual setting, one way was automatic setting.

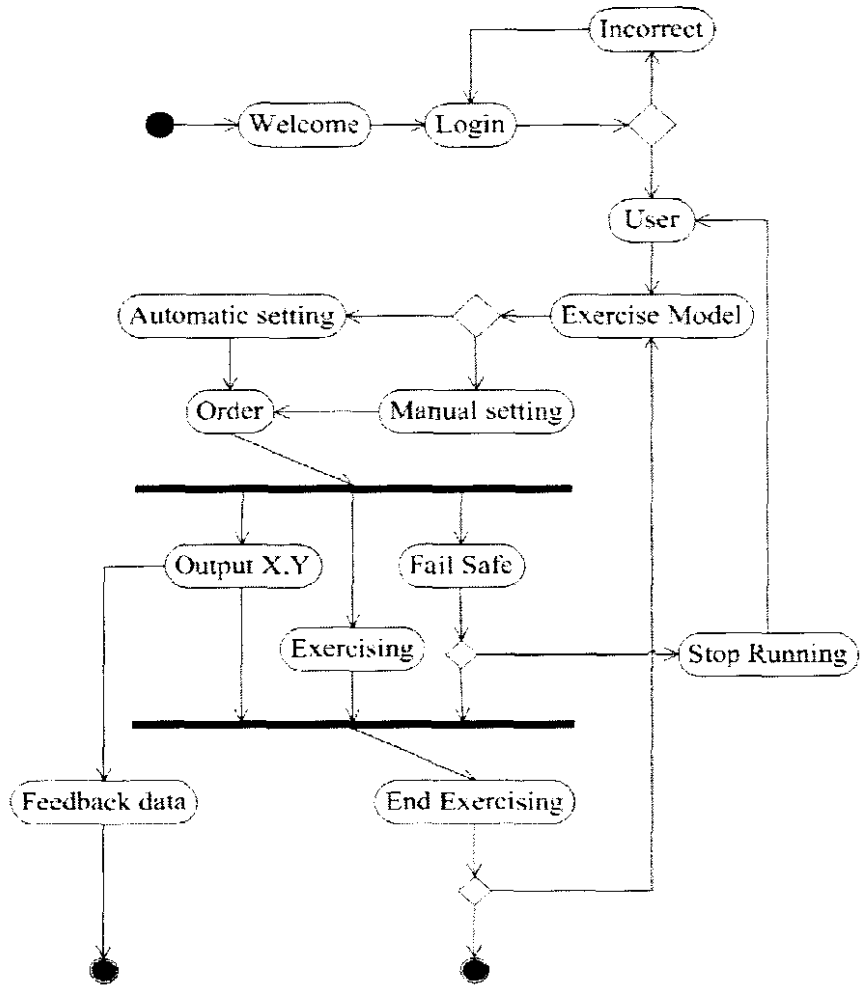


Fig1.2 Exercise activity diagram

The following goals were established:

Develop an Information system

The following outputs were:

1. A function for choosing an exercise mode and intensity for the exerciser;
2. A function for Fail-Safe operation;
3. A database system to store and select information of exerciser;
4. A module to capture data from database;
5. A module to capture the situation of exercise real time;

6. A module to make an exercise plan for next time;
7. A module to set exercise mode and intensity.

The following outcomes were determined:

1. A database system has been developed properly.
2. An information system has been developed properly.

1.5.2 Research activities

Activity 1.1: Develop a module to control a Pneumatic Actuator;

Activity 1.2: Develop a module to capture data from the exercise machine;

Activity 1.3: Develop a module to supply a series movement following appointed mode and intensity;

Activity 1.5: The exercise machine supplies a function for choosing an exercise mode and intensity for the exerciser;

Activity 1.6: The exercise machine supplies a function for Fail-Safe operation;

Activity 2.1: Develop a database system to store and select information of exerciser;

Activity 2.2: Develop a module to capture data from database;

Activity 2.3: Develop a module to capture the situation of exercise real time;

Activity 2.4: Develop a module to make an exercise plan;

Activity 2.5: Develop a module to set exercise mode and intensity.

1.5.3 Target group

Fitness industry

Medical insurance industry

1.5.4 Research resources

1. Finances.
2. Research lab with computers connected to the Internet.
3. Exercise machine can be controlled.
4. Instruments to control the exercise machine from National Instruments.
5. Software include LabView, Microsoft Visual Studio .NET.
6. Knowledge from existing programs.
7. Existing academic and practitioner literature.

1.5.5 Limitation

The current exercise equipment imposes fixed-force exercise on users with the dominance of static equipment designs, and the standard in strength equipment design is based on static force configurations providing single exercise function. These formats have remained relatively standard with minor improvements.

There is not at present a system that controls an exercise machine remotely. So Doctors and Coaches can't supervise exercisers remotely, can't get exercise and fitness data automatically. They must do that face to face. If there were a system that supplied a function that controlled an exercise machine remotely, the exercisers could do exercises at the gym or at home, and Doctors and Coaches could supervise the exerciser and get exercise and fitness data at home or at the office through the Internet.

1.6 Research Timetable for Implementation

This project is divided into six phases, with most of the project activities being part of the third phase:

1. Establishment and clarification
2. Building up the network lab
3. Design and implementation
4. Testing
5. Reporting
6. Publishing

1.7 Organisation of the Thesis

The first section of this research included a literature study, a technology study, and an Internet search to find out what the best practices and user-requirements were. These were used to design, implement and develop information system architecture for an intelligent semantic machine control system. The followed section was implementation of information system in an intelligent semantic machine control system. The dissertation structure is shown below.

In chapter 1, introduction to the study, establishing the field of research, preparing basic concepts for intelligent semantic machine control system and information system architecture were described. The research problem was stated and aims and objectives of the research were described. The end users were defined.

In chapter 2, literature review, the background of intelligent machine, semantic

machine, machine control and information system architecture was provided. The literature review brought the researcher into contact with new ideas and approaches, and revealed unknown data sources.

Methodology, basic concept of Object Oriented Methodology and UML-based design methodology was provided in chapter 3. OO design and UML were deemed suitable for this research project in terms.

In chapter 4, case study, a greater Electronic Fitness Management System was introduced, environment and implementation technology were provided. This analysis will impact greatly on design and implementation of the information system architecture. In section 4.4, this covers the research project implementation in information architecture. In section 4.5 and 4.6, the whole project was tested and evaluated.

Finally, in chapter 5, encapsulating the conclusions and recommendations, three aspects of the research project emerged: in terms of the project itself, functionality; secondly, implementation of the technology that was used; thirdly, the project methodology.

CHAPTER 2 LITERATURE REVIEW

The purpose of this literature review is to provide the reader with general overview of "an intelligent, semantic machine control system" and "information system architecture". This chapter focused on relevant "intelligent, semantic machine control system" concepts and various models of "information system architecture", which constituted the foundation for "the development and implementation of an intelligent, semantic machine control system with specific reference to information system architecture design" in the following chapters. "The intelligent, semantic machine control system" concepts provide a basic understanding and insight into the control system. In fact, they must be known before analyzing any control system, so they were provided in section 2.1 for "intelligent control system", section 2.2 for "semantic in an intelligent control system", section 2.3 for "machine control system". In Section 2.4, the characteristics of the "information system architecture" model as required in this dissertation were discussed.

2.1 Intelligent control system

As mentioned in background, in order to achieve a highly autonomous behaviour for complex systems one can enhance today's control methods using intelligent control systems and techniques.

The area of Intelligent Control is a fusion of a number of research areas in Systems and Control, Computer Science, and Operations Research among others, coming together, merging and expanding in new directions and opening new horizons to address the problems of this challenging and promising area. Intelligent control systems are typically able to perform one or more of the following functions to achieve autonomous behaviour: planning actions at different levels of detail,

emulation of human expert behaviour, learning from past experiences, integrating sensor information, identifying changes that threaten the system behaviour, such as failures, and reacting appropriately. This identifies the areas of Planning and Expert Systems, Fuzzy Systems, Neural Networks, Machine Learning, Multi-sensor Integration, Failure Diagnosis, and Reconfigurable Control, to mention but a few, as existing research areas that are related and important to Intelligent Control. While these techniques provide several key approaches to Intelligent Control, for complex systems they are often interconnected to operate within an architecture which is hierarchical and often distributed. It is for this reason that the areas of hierarchical intelligent control, distributed intelligent control, and architectures for intelligent systems are of significant importance in the design and construction of the overall intelligent controller for complex dynamical systems (Polycarpou n.d.).

2.1.1 Intelligent Control and its usage

The IEEE and the National Institute of Standards and Technology (NIST), in collaboration with other agencies, organized a series of conferences intended to expand the topic of Intelligent Control towards the more consistent, yet more difficult theme of Intelligent Systems starting in 1995, because the problem of intelligent control has turned out to be an intrinsically interdisciplinary one (Meystel A. and Messina E. 2000).

A popular and all-encompassing definition is by Albus (1991):
"... intelligence will be defined as an ability of a system to act appropriately in an uncertain environment, where appropriate action is that which increases the probability of success, and success is the achievement of behavioural subgoals that support the system's ultimate goal."

Pang describes an intelligent controller as a controller that is utilized for shaping the behaviour of an intelligent system (Pang 1991). The distinct properties of this

controller are to provide the following features:

1. It should "know" what actions to take and when to perform them
2. It should reconcile the desirable and feasible actions
3. It should vary the high resolution details of control heuristics
4. The acquired control heuristics should be the most suitable ones and they should change dynamically
5. It should be capable of integrating multiple control heuristics
6. It should dynamically plan the strategic sequence of actions
7. It should be able to reason between domain and control actions. In other words, it should be able to use at least two levels of resolution simultaneously: the level of the domain actions and the domain of the control actions.

In their foreword White and Sofge (1993) wrote, 'intelligent control' should involve intelligence and control theory. It should be based on a serious attempt to understand and replicate phenomena that we have always called 'intelligence' – i.e., the generalized, flexible and adaptive capability that we see in the human brain."

Cai states that intelligent control possesses four features (Cai 1997). These are

1. It is a hybrid control process, containing knowledge of mathematical and nonmathematical models; it has no known single algorithm for dealing with the complexity, incompleteness, and ambiguity it is confronted with.
2. Its core is in the higher level that organizes the problem solving. This higher level of analysis and decision-making and planning requires related technologies, such as symbolic information processing, heuristic programming, knowledge representation, fuzzy logic, and automation of the discovery of similarity amongst the solving processes. "There exists intelligence in the artificial problem- solving process."
3. It is an interdisciplinary field, requiring coordination with, and assistance from, related fields, such as artificial intelligence, cybernetics, systems theory, operations research, etc.

4. It is a research area under development only recently and would experience faster and better development should a better theory for intelligent control be found.

2.1.2 Characteristics of intelligent control

Some definitions of intelligent control would limit it to those systems that rely on soft computing techniques, such as fuzzy logic, neural networks, and genetic algorithms. Another perspective is to analyze intelligent control systems with respect to their characteristics, which were inherent in the quotes of the previous section (Meystel A. and Messina E. 2000).

Looking at control laws, which are the heart of a control system, whether it is intelligent or not is the beginning. An analysis of control laws indicates that they are subservient to the control system's goal of reducing the deviation from a pre-specified trajectory and/or the final state (Maybeck 1979). It is more typical in the present paradigm of automatic control to consider control laws to be part of a broader "control strategy" for the overall system (Song & Mitchell 1993). The assignment for the control law – to keep a certain variable of interest within some bounds around a reference trajectory – comes as the result of some external intelligence operating beyond the limits of the intelligence of the particular control law. Intelligent controllers tend to incorporate these assignments (and their genesis) into the function of the controller. This difference is a fundamental one.

1. In the *fuzzy logic controllers*, the fundamental transformations move the set of input information from the high resolution domain into the low resolution domain by using the tool of "fuzzification." Fuzzification plays the role of a generalization procedure, because it searches for adjacency among information units, focuses attention, and groups. Fuzzy logic controllers contain control mappings only for generalized information at the lower resolution. When the required control action is found, this lower resolution recommendation is

instantiated or moved to the domain of high resolution by the process of "defuzzification". Hence, fuzzy control systems are multi-resolutional, and they move between resolutions for the purposes of achieving the required control performance (Meystel A. and Messina E. 2000).

2. *Neural networks* are a computation device for generalizing in a vicinity (spatial or temporal). They are a natural tool for moving information from higher to lower levels of resolution. Therefore, neural networks, such as described in (Hesselroth et al. 1994), are multiresolutional systems.

3. *Expert System based controllers* make a judgement concerning generalizations and rules to be applied at the lower resolution (Ling 1993). They too presume a multi-level, multi-resolution framework within which they operate.

4. *Hybrid logic control systems* are multi-level control systems in which lower levels of resolution are formulated in terms of logic based controllers, while the higher resolution levels are analytical controllers (e.g., PID, Kalman). Again, there is at minimum a dual-level architecture of control, with differing resolutions.

5. *Behaviour-based controllers* employ a concept of superposition of the activities of multiple controllers working simultaneously, each providing for a separate type of behaviour. A hierarchical structure of some sort generally underlies implementation. A low resolution level may select which behaviour controller is invoked at which time, or it may arbitrate between the different proposed behaviours. Each individual behaviour generation controller generates sub-goals for itself (Meystel A. and Messina E. 2000).

Although the previous examples are not an exhaustive analysis of techniques and approaches for intelligent controllers, they serve to illustrate that certain

generalizations can be made about the essence of what distinguishes a controller that is intelligent from one that isn't. A meta level of control, which functions a lower level of resolution is necessary to guide the underlying control system and expand its envelope of functioning (Meystel A. and Messina E. 2000).

2.1.3 Beyond intelligent control to intelligent systems

Musto and Saridis have defined the intelligent control problem as "intelligent control is postulated as the mathematical problem of finding the right sequence of internal decisions and controls for a system structured according to the principle of increasing intelligence with decreasing precision such that it minimizes its total entropy" (Musto & Saridis 1998).

If intelligent control is responsible for finding the right decisions and controlling the system to enact those decisions, then there must be complementary supporting functions which provide models of the world that the system is functioning in. There must be perception and world modelling processes that occur alongside with the control function (Meystel A. and Messina E. 2000).

Key issues of the domain highlighted in (Lemmon 1994) include

1. "A desirable property of intelligent systems is that they are 'adaptive'...
2. Intelligence is an internal property of the system, not behaviour...
3. A pragmatic reason for focusing on 'intelligent' control systems is that they endow the controlled system with enhanced autonomy..."

2.1.4 How to formalize intelligence in control

Taking a holistic view of an "intelligent system," it can be begin to considered that the formalization of the organization of the system and how its performance is measured. One approach to formalism is via an Elementary Loop of Functioning (ELF), which is

common to most concepts of intelligent systems. Fig 2.1 shows the components of the ELF. ELFs describe a closed loop of functioning common to intelligent systems. A detailed description of the formation of hierarchies of nested loops is described in (Albus & Meystel 1997).

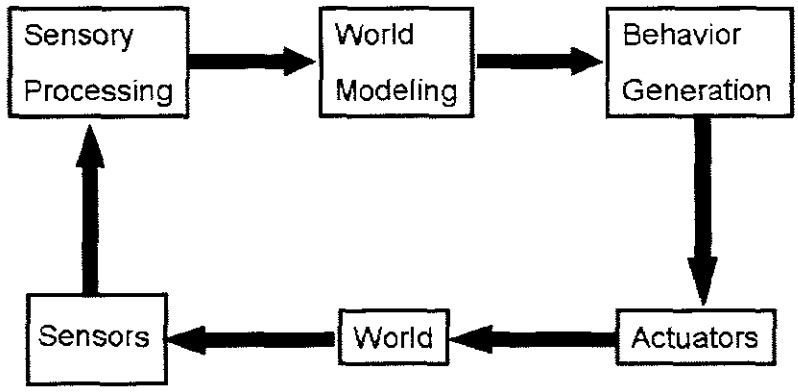


Fig 2.1 Elementary Loop of Functioning (Albus & Meystel 1997)

It is the contention that the emphasis in intelligent control has been placed primarily on the Behaviour Generation elements (and its constituents, such as planners and executors). Broadening the scope to consider all the elements of ELF working together within a hierarchy, we can consider the intelligence of the system as a whole. The construction of a system based on the ELF model is driven by a verbal description of the requirements for its operation – its "story." At this time, transforming the verbal story into the formal ELF is in the stages of exploratory research (Meystel A. and Messina E. 2000).

The domain of application for which we need the intelligent controller must be declared. A distinction is made between closed and open systems. Closed systems can be characterized by clear assignments of the problem to be solved and the ability to construct a complete list of concrete user specifications in terms of measurable variables. On the other hand, in open systems, the problem is not totally clear, its parts are not concrete, the variables are not all listed at the beginning of the design process, and the methods of observation and registering are not outlined. Intelligent systems

based on ELFs are appropriate for the open system problem (Meystel A. and Messina E. 2000).

Looking at the functioning of a system from the outside, we can devise degrees of intelligence, which are linked to the specifications of the controller (Meystel A. and Messina E. 2000).

The necessity for gradations in the measurement of success of the intelligent system leads to gradations in the measurement of the components of the ELF (Meystel A. and Messina E. 2000).

For the world modelling, or knowledge representation, we can look at parameters that are meaningful, such as the competence of the knowledge base (can it answer questions that the rest of the system poses in order to do its job?), the memory depth, the number of information units that can be handled, the number of levels in the system of representation, the number of associative links between units of information, and so on. For Behaviour Generation, we can assess parameters such as the horizon of planning at each level or resolution and the size of spatial scopes of attention. For Sensory Processing, parameters to be considered include the depth of details taken into account during the processes of recognition at a single level of resolution, the number of levels of resolution that should be taken into account during recognition, and the minimum distinguish ability unit in the most accurate scale (Meystel A. and Messina E. 2000).

Further development of the analysis of intelligent controllers requires construction of inner ELFs for each subsystem of the main ELF (see Fig 2.2) (Meystel A. and Messina E. 2000).

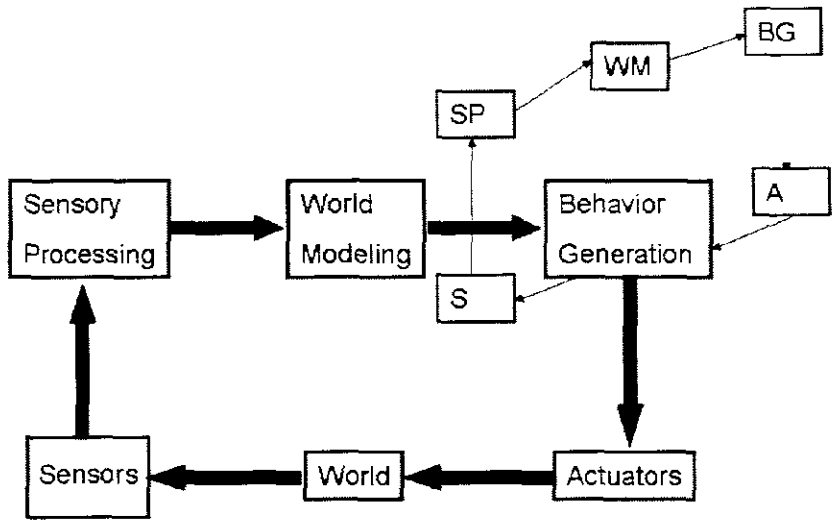


Fig 2.2 ELF_{BG} pertaining to the BG-module of the main ELF (Meystel A. and Messina E. 2000)

2.1.5 Control of a pneumatic servosystem using fuzzy logic

The electro-pneumatic control systems are widely applied in the industrial automation technology, especially, when action is controlled by using programmable logic controllers (Moreno 2000).

Conventionally, when a control must be optimized, we have to build up a mathematical model, (Astrom & Häglund 1999). The control performance of a traditional controller fully depends on the accuracy of a known system dynamic model. The complex pneumatic servodrive positioning process has non-linear and time varying behaviours; thus it is difficult to derive and identify an appropriate dynamic model for traditional controllers, (Ming-Chang & Tsung-Wei 1998). Then the control system should be analyzed and a controller can be designed. It is very difficult to get an accurate and linearised mathematical model; therefore the fuzzy control technology is applied.

As is well known, Zadeh, (Zimmermann 1991), originated the fuzzy theory in 1965.

In recent years, the control technology has been well developed and has become one of the most successful tools in the industry. In fact, nowadays, there are several programmable logic controller dealers who offer fuzzy control tools in their product.

This kind of control has been applied in the industrial world (process and automation) giving an optimal effect, (Klein 1993). The results of this application have demonstrated that the fuzzy control shows better benefits in comparison with those offered by other PID controllers. The main advantages of the fuzzy controller are these, (Sorli et al. 1999):

1. It is not necessary to build a detailed mathematical model. Despite this, in the present work, this point is considered by means of Bond Graph Technique.
2. The fuzzy controllers have a high strength and a high adjustment.
3. They can operate with a high input number.
4. They can be adapted easily into non-linear systems.
5. The human knowledge can be easily applied.
6. The process development time is relatively lower.

The essence of fuzzy logic control is that appropriate linguistic fuzzy rules are chosen, using some decision-making process, from a rule table constructed using human control experience and databases. In the present work the fuzzy rules are established by trial and error with the concept of symmetry. Fuzzy set theory is employed to simulate the logic reasoning of human beings (Moreno 2000).

Fuzzy control has been demonstrated to provide highly satisfactory results in terms of accuracy, repeatability and insensitivity to changes in operating conditions, (Ferraresi, Raparelli & Velardocchia 1990). Classic controls satisfies the requirements for stability, accuracy and rapid response; providing that there is an optimal match between the real values of the system's physical parameters and the values used for control design, and there is no external interference (change in load). Advanced control techniques (e.g. optimum, robust, self tuning) require highly sophisticated and

complex control algorithms if they are to be any effective in use (Moreno 2000).

Applying fuzzy control to a continuous pneumatic positioning system is particularly advantageous in terms of simplicity of design and implementation, and thus significantly reduces the time required to develop the entire system, (Ferraresi , Giraudo & Quaglia 1994). Experience shows that the success of a fuzzy control depends on the level of knowledge concerning the positioner's physical behaviour (Moreno 2000).

LAYOUT OF THE EXPERIMENTAL DEVICES

The functional pneumatic positioning system with fuzzy control is shown in Fig2.3. The actuation element consists of a double-acting pneumatic rodless cylinder. The cylinder is a Heavy Duty model of Norgren - Herion. The cylinder chambers are connected to a flow rate proportional valve, the ISO 1 Isotron model of Norgren (1000 Nl/min). The flow proportional valve has an internal position closed loop, and a cut-off frequency of 80 Hz.. The system incorporates provision for varying the inertial load acting on the actuator bore (Moreno 2000).

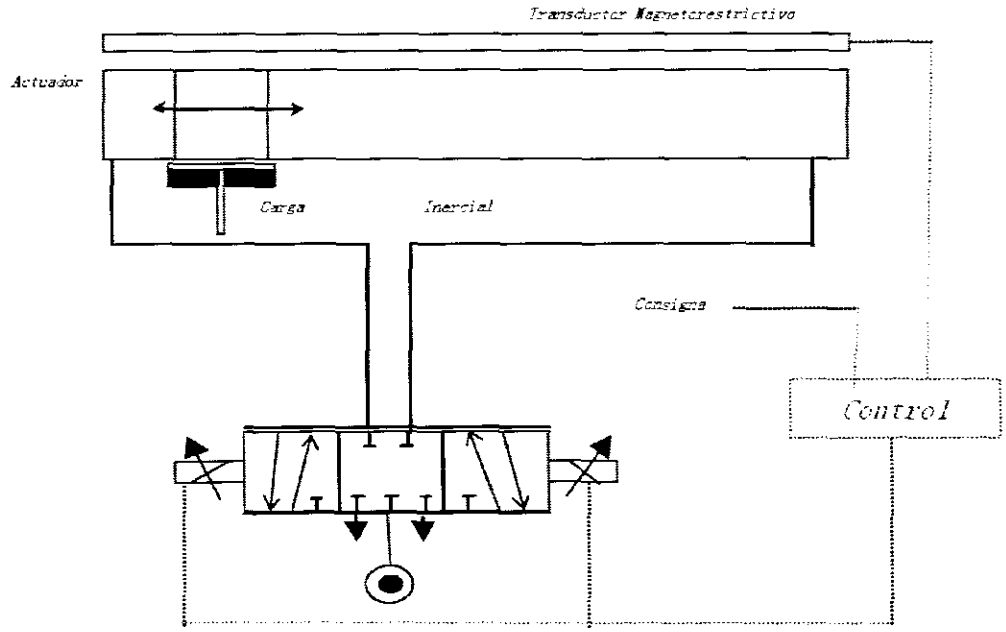


Fig 2.3 Positioning System (Moreno 2000)

A magneto restrictive analog sensor of displacement closes the loop in terms of position; this sensor can give the speed signal also offering a high resolution. The Fuzzy Controller generates valve control signals by processing the resident program on the basis of measured magnitudes. The fuzzy system consists of a National Instruments Data Acquisition Card (model PCI 1200), the control program is run under Labview, and Labview Fuzzy Toolbox is also used. For the control system, an electrical cupboard has been built (Moreno 2000).

The rule sets and the membership functions for input and output variables are transferred to the fuzzy processor. Rules and functions are written using the National Instruments Fuzzy Logic Toolbox. The fuzzy inference process used by the controller is the min-max- centre of gravity method (Moreno 2000).

SYSTEM CHARACTERISATION

As explained before, the positioning system basically consists of a pneumatic actuator operated by a servovalve with a position closed-loop. Pneumatic actuator performance can be determined at first approximation by steady state tests. The real working conditions including coupling between actuator and proportional valve, cannot be completely achieved if the design specifications require reduced motion times. Coupled system characteristics can only be evaluated by performing a dynamic analysis of the system, specially dealing with short stroke heavy loaded actuator or long stroke low loaded actuator (Moreno 2000).

Moreno (2000) claimed in his study that assuming the type of actuator, single or double acting, with stiff or flexible wall and supposing that the power regulation element has been chosen, the parameters that influence the dynamics of the actuator are:

1. The sizes and geometry of the chambers, defined by the linear or rotary stroke, bore and rod, in the linear actuators.
2. The type of seals and the mobile support, which exert a strong influence on friction forces.
3. The size of the regulation valve.
4. The length and dimension of feeding pipes.
5. The magnitude to be controlled by the actuator (force, position or velocity).
6. The level of supply pressures.
7. The entity of disturbances related to the controlled magnitude.

The above mentioned parameters interact in a very strong non-linear way. The servo system performances also strongly depend on the control strategy.

2.2 Semantic in an intelligent control system

Much research on intelligent systems has concentrated on low level mechanisms or limited subsystems. We need to understand how to assemble the components in an architecture for a complete agent with its own mind, driven by its own desires. A mind is a self-modifying control system, with a hierarchy of levels of control, and a different hierarchy of levels of implementation. Only when we have a good theory of actual and possible architectures can we solve old problems about the concept of mind and causal roles of desires, beliefs, intentions, etc. The global information level "virtual machine" architecture is more relevant to this than detailed mechanisms. E.g. differences between connectionist and symbolic implementations may be of minor importance. An architecture provides a framework for systematically generating concepts of possible states and processes. Lacking this, philosophers cannot provide good analyses of concepts, psychologists and biologists cannot specify what they are trying to explain or explain it, and psychotherapists and educationalists are left groping with ill understood problems. The paper outlines some requirements for such

architectures showing the importance of an idea shared between engineers and philosophers: the concept of "semantic information" (Sloman 1994).

2.2.1 Semantic information

The semantic concept of information, used informally by engineers, is closer to the familiar, though extremely ill-defined, notion of the "meaning", or "information" that may be conveyed by a message, or stored in a book or videotape. It is applicable to internal states as well as external communications. This includes ordinary concepts of mental states and processes such as "understanding", "believing", "desiring", "deliberating", "perceiving", "regretting", and many more. Philosophers call these states and processes "intentional" because they refer to something outside themselves, including, in some cases, nonexistent entities, e.g. hallucinating daggers, intending to perform actions which turn out to be impossible, or praying. Doing, preventing and trying are also semantic states, for they involve reference to future results (Sloman 1994).

All semantic information-processing depends ultimately on (internal) syntactic capabilities, for semantic information-processing depends on syntax-processing (i.e. structure-manipulating) engines, whether in computers, neural nets or other mechanisms. Agents need representational forms whose syntactic manipulation is semantically useful, unlike "languages" invented merely to illustrate the theory of syntax or parsing. A notation some of whose syntactically well formed formulae are semantically uninterpretable (like "Happiness eats democratic numbers intelligibly") is wasteful, though some meaning gaps may be important seeds of semantic development. Internal syntax is useful for processing information if it provides syntactic manipulations that map onto useful semantic relationships (e.g. syntactic derivability preserves truth). Much of the development of science and mathematics has been the creation of new formalisms that usefully compile semantic relationships into syntactic ones (Sloman 1994).

Computer programs have several kinds of semantics. One sort (called the "information level" below) refers to the application domain, usually outside the computer, e.g. information about employees. Another sort refers to the abstract data structures manipulated at run time in the machine, e.g. numbers, strings, lists. Another refers to the low level machine instructions and memory locations manipulated when a program runs. Some languages also include compile-time semantics e.g. for macros, compiler directives or type definitions, which control compilation rather than subsequent running. Semantic information can vary both in content and pragmatic role. For instance, it may be about what is the case, about what to do, about how to do things, or it may express a question or test. Which pragmatic roles are possible depends on the architecture (Sloman 1994).

Objects with semantic states include:

(a) Humans and other animals

(b) Passive artefacts like books

(c) Active artifacts

Such as factory control systems, office automation systems, are often more like (a) than like (b). Questions about the semantic states of biological control systems, information in DNA, chemical or neural messages, are also important, but will be ignored (Sloman 1994).

2.2.2 Semantics in control systems

Engineers designing plant control systems, office support systems and the like, often start at the global information level, analyzing semantic requirements for the whole system. For example, a system may need information about the environment, rules and procedures to be followed, legal constraints, company objectives and which risks to avoid. Meta-information (information about information) is also needed, for example: where to get certain information, what to do when it arrives, how to cope

with contradictory reports, and so on. Internal monitoring might yield meta-information about how quickly information is dealt with, which kinds turn out to be unreliable, and so on (Sloman 1994).

Sloman (1994) claimed in his study that designing information systems also raises implementation issues at different levels, such as:

1. how information will be stored in the system
2. how to access it
3. selecting forms of representation
4. selecting syntactic transformations
5. selecting programming language(s) and operating systems
6. selecting computers, interfaces, network links, etc
7. functional decomposition of the system

The semantic, or information level, specification, e.g. that the system must include information about employees and their roles and use it to perform certain tasks, says little about such implementation details. The specification can be given in an implementation independent way: including requirements for and the behaviour of a certain kind of information-processing virtual machine, leaving the computational or electronic details concerning "lower level" virtual or physical machines for later (Sloman 1994).

Moreover, implementation details may be revised as technology advances. Processors used, the memory technology, and even the programming languages and some of the low level algorithms may all change without implying any change in what information is processed, as far as the users and designers are concerned: i.e. the global information level description is not affected Sloman 1994).

However, information level descriptions may imply a certain sort of architecture: a top level functional decomposition, defining which sorts of major subsystems coexist,

and which information they handle. For instance, being undecided about whether to go to A or to B, presupposes mechanisms for manipulating goals, for evaluating and selecting between alternatives, and for acting on a selection (Sloman 1994).

For subsystems we can also define an information level: what they "know" about the rest of the system, i.e. their environments and their tasks. Typically, users will not be concerned with that, though designers and maintainers will (Sloman 1994).

2.2.3 Implementing information states

Our understanding of how one machine "implements" another is still mostly intuitive, for we lack good general theories and terminology. Nevertheless, there is no mystery: we can make it happen (Sloman 1994).

A working system always has a physical level of description. Phenomena at other levels may be "emergent" in that concepts needed to describe them cannot be defined in terms of the lower levels, and the laws of behaviour of higher level virtual machines are different from and not derivable from the laws governing the lower implementation levels. Information level concepts like "refers to" or "attempts to", or, for that matter, "customer", cannot be defined in terms of mass, velocity, voltage, etc., and neither can the information level principles of behaviour be derived from physical laws: e.g. they may depend on business practices, legal constraints and the social environment, which change while physics doesn't. Mental states may depend on a culture, e.g. physics doesn't determine musical style or acceptable grammar. Thus information level specifications enrich our ontology with new concepts and new laws, relative to physics (Sloman 1994).

Several levels of abstract "virtual machines" can coexist in a single system, each with their own information level characteristics. In a word-processor one abstract machine performs operations on chapters, sections, paragraphs, words, footnotes, and so on. It

may be "implemented" in terms of another that manipulates arrays, strings, lists, and other data structures, corresponding to "high level" programming languages. Below that level is a virtual machine defined in terms of operations on bit-patterns, describable in the computer's "machine code". Still lower levels consist of abstract digital machines, described in circuit diagrams. Below that are physical states of components, describable in the language of physics (Sloman 1994).

There is no well defined "bottom" level. The lowest level actually considered depends on the task. For many software engineers lower layers are irrelevant (except when things go wrong), though for compiler writers and hardware designers they are crucial. Sometimes close-coupling between high and low levels is useful, e.g. microcode instructions invoking high level procedures, quantum-based randomizers, and perhaps chemical soups for global control in future computers (Sloman 1994).

Normally concepts at each level are not definable in terms of the concepts at a lower level. Word-processor concepts, like "page" or "sentence" are not definable in terms of concepts of physics, nor in terms of arrays, lists or strings. Page-formatting rules are not deducible from physics, nor from equations defining data-types. Each level defines an emergent ontology, with its own laws (Sloman 1994).

2.2.4 Implementation and supervenience

Many have likened the relationship between an information level virtual machine and lower levels in a software system to the relationship between mental processes and the physical brain mechanisms implementing them. Mental descriptions, like "believes" and "desires", are used in ignorance of implementation details, just as information level descriptions of software systems are usable by people who know nothing about the programming languages, data structures and algorithms, or underlying electronic mechanisms (Sloman 1994).

Some information states involve relationships to external objects and therefore cannot be implemented solely in terms of internal states. A computing system cannot store information about Joe Smith solely in virtue of internal states: how the system is related to that external individual also matters. Similarly my beliefs about Joe cannot be implemented entirely in my brain states. Such information level states have "intrinsically relational content", and the environment provides part of their implementation. (Some philosophers label this "wide content" or "broad content".) Thus two physically identical systems in different locations will not necessarily contain absolutely identical information (Sloman 1994).

Particular lower level states are not necessary for the high level states, if alternative implementations are possible. Neither are they sufficient, if successful implementation depends crucially on the current environment, like reference to Joe (Sloman 1994).

In philosophical terminology, mental phenomena are "supervenient" on physiological or physical phenomena. Similarly, computer-based information states are supervenient on physical phenomena (both internal and external). This philosophers' relation is simply the converse of the engineer's relation of implementation. (Both need to accommodate intrinsically relational content.) Philosophers grappling with "emergence", or "supervenience" (e.g. Robinson 1990, Horgan 1993) might be helped by software engineering examples which are already well understood (Sloman 1994).

Animal or machine control systems, including human minds, have an underlying physical implementation, whether based on transistors, neurons, or rotating wheels. But this does not imply that different items of information must be mapped onto physically separable components (physical symbols). High level virtual machine details (including semantic content) need not correlate with or map onto physical structures. Distinct items of information can be superimposed on electromagnetic waves and later separated out using filters. Neural nets allow information items to be

superimposed and distributed over a collection of synaptic weights, or over activation levels of a set of neurons. Two numbers can encode the row, column and both diagonals a chess piece is on – four items of information superimposed on two. Huge 'sparse' arrays in a virtual machine include far more components than the physical structures that implement them. A set of axioms can encode, in a distributed and superimposed form, an indefinite variety of different items of logically derivable information, and which information is extractable can vary with context or time-pressure. Finally, relationships with the environment may help to determine content (Sloman 1994).

In such cases, examination of internal physical structures will not reveal information systems they implement: for much of it is implicit in how substructures are used by other components (Sloman 1994).

Some implementation details have little impact on overall functionality: they make a marginal difference (e.g. to speed, or to reliability under high temperatures that rarely occur) or a big difference only in rare situations (e.g. getting most common questions right). Whether particular subsets of an architecture use neural nets or some other mechanism may be marginal in relation to the functioning of the whole system. It is global design that matters most: which sub-functions are provided, how they are linked, how they are used and how they change. Architecture dominates mechanism (Sloman 1994).

2.2.5 Information and control

What is special about information-processing systems? Do trees and clouds store and use semantic information? Are they understanders, and if not why not? Nevertheless we can make a rough distinction between mechanisms controlled only by physical laws and mechanisms controlled by virtual machines involving information. The latter involves explicit prior representation of possible actions prior to their execution

(Sloman 1994).

Examples are systems containing two or more control sub states capable of producing different behaviour (internal or external) between which selections are made, for example by examining some other sub state of the system. Simple examples include computing systems that support instructions like:

if <condition> then <action1> else <action2>

In more sophisticated cases the <actions> are parameterised, with details filled in as a side-effect of testing a <condition>. (That usually requires lower level conditionals to have explicit alternatives.) What is then explicit initially is a schematic action specification, with building blocks for creating the final specification in advance of performing the action. By contrast, movement of a cloud or a tree depends on external and internal conditions but is not controlled by selections from or construction of explicit prior representations of options (Sloman 1994).

A slight generalisation accommodates connectionist systems: they exhibit parallel and cumulative conditional influences, and some depart from simple binary logic. The pattern of firing of a set of neurons typically depends on

1. The previous pattern in some other (possibly overlapping) set
2. The current weights on the connections between the two sets
3. A decision function for combining weighted influences

This is also conditional control, but all conditions are tested in parallel and the outcomes are sent in parallel to representations of possible actions, which are triggered or deactivated on the basis of the total accumulated support or inhibition they receive. Such networks may be clocked or asynchronous, based on discrete or continuous variation (e.g. of weights or activation levels), and may use different numbers of coexisting layers and connection topologies (e.g. cyclic or acyclic). These are all variants of the general notion of information-based control, as are probabilistic or fuzzy logic systems. However, we should not expect a sharp and total division between systems that are controlled by information and those that are not. This is

another "cluster concept" (Sloman 1994).

Among the useful features of information-based control is context sensitivity i.e. producing different behaviours on different occasions in response to the same specific (internal or external) stimulus, because links between cause and effect are indirect insofar as they depend on conditionals (Sloman 1994).

How a condition works may change conditionally under hierarchic control. How A influences B can depend on information structure C. How it depends on C can depend on D (e.g. if D changes the tests in C, rearranges the order of testing or alters connection weights). The exact role played by sub states of C, and how the information is used to change B, can vary enormously from moment to moment, under the influence of other information states. How D influences the testing can depend on E, and E's influence on D may be modified by F, and so on. For instance the length of time a program P runs can depend on the scheduler S, and which files P may access depends on the file manager, M, where both S and M can be modified by an administrative tool, or may even be modified by the operating system itself using performance statistics (Sloman 1994).

In contrast with animal brains, artificial information-based control mechanisms are well understood. In simple cases programs access information stored in memory, or in input registers, and what they find determines selection of actions. A more abstract virtual machine in a vision system may at one instant interpret an intensity discontinuity in an image array as a crack in a surface, then moments later as an edge of a surface or as a stretched string. The resulting actions will be different (Sloman 1994).

In such cases, how A affects B is not a physical law. Compared with control by physical (e.g. mechanical or electrical) influences, information-based control:

1. Is more flexible (e.g. Can be more sensitive to changing context)

2. Admits more rapid change of control relationships
3. Allows more superimposed layers of dispositions concerned with different functions with different time scales
4. Permits effects to be saved up for future use (so that feedback loops need not have fixed time constants)
5. Supports teleology of a kind once thought mysterious: namely control by representations of future objects that don't yet exist, including pipedreams that never will

These features depend on different internal states having different control functions and embodying different sorts of information. This functional differentiation of control states is what I call (high level) architecture. In general it will not map onto physical architecture in any simple way. That is one reason why the evolution of biological control capabilities is so hard to study: virtual machines leave no fossil records (Sloman 1994).

2.2.6 Concluding comments

The intuitive concept of semantic information or "aboutness" is part of common sense and also philosophy and engineering. It applies to information states and processes as opposed to physical and physiological states and processes. Information states enter into causal interactions and control functions in high level virtual machines, enabling sophisticated and flexible internal and external behaviour (Sloman 1994).

We currently understand a lot about causal connections in information level virtual machines in computing systems. This provides clues about how to think about causal connections in the high level virtual machines constituting human minds, and may eventually lead to an improved theory-based classification of mental states, just as understanding the architecture of matter gave us improved concepts of kinds of stuff. E.g. our muddled ideas about "consciousness" including the prejudice that "conscious states" are intrinsically different from others may prove merely to rest on a distinction

between states that are and states that are not accessible by certain high level self-monitoring processes(Sloman 1978). Some states and processes that are inaccessible to consciousness may be exactly like accessible ones, and the boundary can change through training (Sloman 1994).

Many unanswered questions remain. E.g. how and why did different information-processing capabilities evolve? This breaks down into several questions. How did different forms of syntax evolve? How did different functional roles (pragmatics) for control sub states evolve? How did different semantic capabilities evolve? How did different inference capabilities evolve? And how did architectures evolve that integrate all these capabilities in a multifunctional whole? Whether computer-based systems could replicate high level functionality of natural brains is also open (Sloman 1994).

Answering these questions requires AI theorists to survey and classify the variety of information-bearing states to be found in human beings, other animals, and machines of various sorts. We need to study "dimensions of sophistication" in which architectures can differ, including: the number and variety of concurrent high level functions, the variety and complexity of forms of syntax and structure manipulation used in information stores and control states, the flexibility and depth of perceptual processing, the variety of sources of motivation, different kinds of internal self monitoring, different kinds of self control and internal management, varieties of self-modification and learning, including modifications of the architecture itself by addition of new capabilities or mechanisms and new links between old ones. These themes are developed in papers listed below (Sloman 1994).

This "exploration of design space" has barely begun, even though some AI researchers prematurely commit themselves to particular representations, e.g. logical or connectionist, or to restrictive architectures (Sloman 1994).

servo systems will have better prospect of application in the future (Gao & Feng 2004).

The nonlinear characteristics, especially the nonlinear friction and the thermodynamic characteristics of the pressure air in the chambers of the cylinder have a bad influence on control accuracy of the displacement controlling of the cylinder. All these nonlinear characteristics depend on the quality of manufacturing process of the cylinder without piston rods as well as on the sealing mechanism such as the band sealing and split sealing cylinder. In addition, there is a series of nonlinear and time-varying factors such as load force, temperature, position of the piston, staying-time at still and wearing out during working procedure. Some attempts have been made to obtain performance of quickness, accuracy and displacement controlling without overshoot by means of conventional controllers based on the theoretical model of the pneumatic servo system. The efforts do not result in any satisfactory results. One of the ways to relieve the nonlinear characteristics of pneumatic cylinders is to utilize precise cylinders with less friction. But this way will increase the cost of building a pneumatic servo system. The advantages of pneumatic systems cannot display. Because there are many influence factors on the control qualities, it is difficult to describe the dynamic performance of the servo system or to build the mathematical model. It is relatively easier to make use of a fuzzy logic controller, even though there are difficulties in designing a fuzzy logic controller and in determining the parameters of the fuzzy logic controller by trial and test (Gao & Feng 2004).

Berger (1994) presented the idea of designing fuzzy controllers according to classification of the controlled systems. The design of fuzzy controller is similar to the design of conventional controllers such as PID, PI and P controllers. Thus the fuzzy controllers can be classified as Fuzzy-PID, Fuzzy-PI and Fuzzy-PD controllers. They

Because the tasks are so difficult, AI may look like a "dead end" to those who do not understand the variety of important but difficult problems that remain. A subject with so much important unfinished business cannot be at a dead end. But we should not insist on some narrow set of explanatory tools and concepts. That would be as silly as trying to restrict physics to the concepts and mathematics that Newton understood. Infant disciplines must be allowed to grow (Sloman 1994).

Many philosophical views are based on emotional commitments and cannot be changed by evidence and rational discussion. In the long term, results of design-based investigations may convince some doubters, but never all. For some, theoretical analysis will suffice. Some will change only after developing personal relations with intelligent androids. Others may be convinced when the new theories help us solve difficult human problems, for example in education and therapy, which, at present are neither science nor engineering but often hit-and-miss craft activities (Sloman 1994).

2.3 Machine control system

For a great demand for servo control systems in the field of manufacturing and industrial processing control, the commonly used systems are electric servo systems, hydraulic servo systems and pneumatic servo systems. Generally electric and hydraulic servo systems are used to meet the need of higher performance with higher costs. The pneumatic servo systems are normally used to meet the need of lower performance because they have the advantages of lower cost, of long working life, of working overload, of anti-explosion and without need of maintenance. But on the other hand, the control accuracy is affected badly by its nonlinear characteristics. Fortunately the application of a fuzzy procedure in conventional regulators is a very simple way to improve the performance of a conventional control strategy. If some robust and adaptive controllers can be found to compensate the influence of the nonlinear characteristics, the control accuracy will be enhanced and the pneumatic

have similar transform performance to the conventional PID controllers.

Berger, Schwarz, and Behmenburg (1995) presented the idea of standard fuzzy controllers, which should not demand that designers have a deep knowledge of fuzzy logic. It makes sure that a suitable fuzzy controller can be chosen to suit a certain type of system. The system can be classified according to the significant dynamic characteristics. The classification of systems can be realized easily by means of the form of step response of the analyzed system in open loop. Corresponding to the type of system, a type of fuzzy controller should be chosen to suit the analyzed system, only if the free chosen scaling factors can be adjusted suitably to be adaptive for the system.

2.3.1 Force control

Force control is a technology that has developed to fill a void in the automated manufacturing process. In many manufacturing processes parts are brought to a net dimensional shape by machining, casting, forging, or molding. These parts often meet the dimensional specifications, but still require additional processing to achieve a desired surface finish. In the case of machining operations, residual marks and scallops are removed from the part. Other processes, such as injection molding, casting, and forging, require the removal of flashing, gates, and parting lines. These blending and finishing operations are applications where switching to a force controlled process rather than a dimension driven position controlled manufacturing process is required (Erlbacher 1995).

Blending of parting lines or removal of cutter mismatch, scallops and flashing requires a human touch. This human touch is a form of compliance and is a property that rigid, position based machine tools do not have. Compliance in the context of automated surface finishing is the ability to compensate for mismatch between a tool and a part surface, and is based on maintaining contact rather than position. The

primary consideration when contacting a part surface is controlling the amount of force being applied by the tool. In automated surface finishing the tool is often an abrasive medium and the amount of force applied directly affects the Material Removal Rate (MRR). The MRR is the amount of material in volume removed in a specific time (Erlbacher 1995).

In general, to successfully implement an automated surface finishing system some type of compliance and force control is needed. There are two commercially accepted methods of force control used in automated surface finishing today. The first method, "through-the-arm" force control, applies force using the position of all the robot axes in unison. The second method, "around-the-arm" force control, uses the robot for positioning motion only, and applies a controlled force through an auxiliary compliant end-of-arm tool (Erlbacher 1995).

2.3.2 Through-the-arm Force Control

Through-the-arm force control has been called the "Elegant Solution" since it appears to be such a simple and natural extension of robot control technology. The appeal of through-the-arm force control results from the compelling urge to draw an analogy between robots and human beings. If a human can apply a force and move a tool over a part, then why not use a robot to move a tool over the same part. A human being has a brain; a robot has a computer controller. A human being has muscles, arms and hands to grasp tools; a robot has servo motors, a manipulator and a gripper. It seems obvious that if a robot were given a sense of touch with some type of force transducer then the machine should be able to apply forces like a human. While this analogy seems obvious there are subtleties which make through-the-arm force control very difficult to successfully implement (Erlbacher 1995).

2.3.3 Around-the-arm Force Control

Around-the-arm force control is a method that is based on using the robot arm for positioning and motion only. This method de-couples the force control from the robot controller and servomotors. The force device can be stationary or mounted on the robot. The robot mounted force control device is an auxiliary tool that adds an additional axis of motion to the system. This axis of motion provides the compliance that is necessary for automated surface finishing. Compliance is extremely important in compensating for part location mismatch or unexpected additional material. The axis of motion or compliance can be either linear or rotary. The linear stroke is usually around 1 inch and the rotary motion is typically 5 degrees. Greater compliant strokes are possible, but not normally used since the robot can easily position the tool within these limits (Erlbacher 1995).

There are two methods used to implement around-the-arm force control: passive and active. Passive force control is an open loop control system with no mechanism to adjust for force errors. Active force control is a closed loop control system that can adjust to reduce force errors. A practical example will help to illustrate the difference between open loop and closed loop systems (Erlbacher 1995).

2.3.4 Passive mechanically counterbalanced force devices

The most common passive devices are the mechanically counterbalanced and the air counterbalanced designs. The first type of device to be discussed is the mechanically counterbalanced unit. The most notable feature of this configuration is the use of a mass to physically counteract the tool weight vector. The counterweight eliminates the need to determine the orientation of the device as it is moved around. After the tooling is mounted to the carriage, metal plates are added or removed from the counterweight until the device is balanced. No matter what orientation the device is

placed in the two masses will counteract each other. In fact a single acting actuator can be used on the mechanically counterbalanced device since it must only apply a force in one direction (Erlbacher 1995).

2.3.5 Passive air counterbalanced force device

The passive air counterbalanced force device differs from the mechanically counterbalanced device by not requiring a mass to offset the tool weight. Adjusting the air pressure on each side of a pneumatic piston actuator compensates for the tool weight. Since the passive air counterbalanced tool can be placed in any orientation by the robot a double acting actuator is required to offset the tool weight (Erlbacher 1995).

Three methods are used to determine the amount of pressure required in the actuator to provide the necessary counterbalancing and the required force output. The first method assumes that the end-effecter will remain in the same orientation relative to the weight vector during part processing. If this is the case then the pressures in the actuator will remain constant throughout the process. To set the pressures in the actuator some type of force measurement system (i.e., a load cell, calibrated weight) is attached to the device. Using the measurement system the regulators are set to produce the required force. The pressure regulators can be manually adjustable with this method, which means that no electrical signals are required. Manually setting the regulators is a very simple solution, but one that provides very little process flexibility (Erlbacher 1995).

The second method for setting the pressure regulators is necessary if the device will not be in a constant orientation. As the orientation changes the pressures in the actuator must change to counterbalance the weight vector and maintain the desired force. When the robot path across a part is taught, the actuator pressures are also set for each point and saved in the robot controller by the user. Adjusting the regulators

requires some type of electronic interface between the robot controller and the force device. The second method can become tedious since each time a new path point is taught, a force measurement system must be used to determine the settings for the pressures in the actuator. During processing of the part the pressure regulating devices will receive new signals from the robot controller each time it comes to a taught point. This means that the accuracy of the force device is directly related to the amount of points that the user decides to teach. If the process path across the part causes the orientation to change gradually then the accuracy may be sufficient. For parts with rapidly changing orientations the force accuracy probably will not be acceptable for demanding processes. This second method is best suited for simple parts with relatively flat features (Erlbacher 1995).

The third method will work with a constant or changing orientation and involves a set of equations that continuously run on the robot controller to adjust the actuator pressures (Erlbacher 1995).

2.3.6 Active force control devices

Active force control differs from passive force control by utilizing a stand-alone controller to manage a closed-loop system that continuously monitors the applied force and corrects for any errors. The active technology is based on directly reading the force applied by the actuator. This guarantees a much more accurate system than the passive method that indirectly applies force based on pressure measurements (Erlbacher 1995).

2.4 Information system architecture

Organizing functionality and content into a structure that people are able to navigate intuitively doesn't happen by chance. Organizations must recognize the importance of

information architecture or else they run the risk of creating great content and functionality that no one can ever find.

2.4.1 Definition of information architecture

At its most basic, Information Architecture is a field and approach to designing clear, understandable communications by giving care to structure, context, and presentation of data and information (Poggio n.d.). It is the term used to describe the structure of a system, i.e. the way information is grouped, the navigation methods and terminology used within the system (Barker n.d.). It is the process of developing the information structure of the system, which takes place during the Define phase, with the aim of creating an effective system and a good user experience. Outputs of this process normally include the system plan, wire frames and parts of the functional specification (Public Life n.d.). It is the art and science of structuring knowledge, and defining user interactions (Brainboost n.d.).

The most common problem with information architectures is that they simply mimic a company's organizational structure. Although this can often appear logical and an easy solution for those involved in defining the architecture, people using systems (even intranets) often don't know or think in terms of organizational structure when trying to find information (Barker n.d.).

In a library, for example, information architecture is a combination of the catalog system and the physical design of the building that holds the books. On the Web, information architecture is a combination of organizing a site's content into categories and creating an interface to support those categories. It stems from traditional architecture, which is made up of architectural programming and architectural planning (Brainboost n.d.).

Traditional architectural programming

The traditional discipline of architecture, which is the design of buildings and physical space, involves problem-making and problem-solving. It requires a thoughtful analysis (programming) to manifest a thoughtful synthesis (design) (Kimen 2003).

Architectural programming is an objective approach to understanding the nature of the task so that a specific problem can be identified as something for space planners and designers to solve. The programmer establishes goals, collects and analyzes facts, uncovers and tests concepts, determines needs, and states the problem. The programmer's responsibilities include: client interviews, research and understanding of emerging technologies, reviews of case studies, budget planning, scheduling long-term deadlines, anticipating the future, and formulating functional requirements. The research results in a program document that specifically outlines the limits of the project and any unique problems (Kimen 2003).

Traditional architectural planning

Between the analysis and synthesis stages exists what William Pena, author of Problem Seeking: An Architectural Programming Primer, calls the synthesis gap. In large projects, a space planner manages this gap by taking the program document and defining the space to be designed, aligning the rooms, and assigning priorities to the interior structural elements. The space planner works with both the programmer and the designer to develop a structure that accommodates the function as well as the form. (Although sometimes, depending on the firm, the space planner is also the programmer or designer) (Kimen 2003).

In information system design, a person who helps develop programs and also plans is an information architect. The information architect maps the entire structure of the system and organizes the positioning of components within sections, developing a

functional and intuitive plan to get the user from point A to point B on the path of least resistance (Kimen 2003).

2.4.2 Fitting information architecture into an information system

Some software design firms have highly compartmentalized departments that separate problem finders from problem planners and problem synthesizers, but flexibility is the key to success. Information architects should meet with clients to help define a project's scope, as well as plot the path to meet the objective and work with the designers and technologists to develop engaging and intuitive visual interfaces. It is important for them to be present during all three phases and to get a client's objectives firsthand. Poor second-hand interpretations can be a project's death. It isn't that managers are inept at translating clients desires, but architects have special architectural questions that a business manager or producer might not be able to intuit (Kimen 2003).

It's also important for information architects to work closely with visual designers, helping to maintain the balance between form and function. Design effects architecture as much as architecture effects design. Working in a vacuum of compartmentalized skills isn't good for anyone, and it's definitely not good for the end result. Information architects also bridge architecture with development and work with technologists, database engineers, and coders (Kimen 2003).

Most of the larger software firms have established IA departments operating under various names. Some firms base their definitions on information system design, while others take a more traditional, physical structure architecture approach. It's impossible to say what works best, because it's relative to the overall environment and work process. In general, it's good to take elements of information system design, library science, traditional architecture, and industrial design and sift through for the elements that most apply to information system design and its nuances (Kimen 2003).

2.4.3 What do architects create for clients?

If there were a template or system for what information architects need to prepare, no one would need them. While there are certain key deliverables that most projects require, the work is most often determined on a case-by-case basis dependent on scope and function (Kimen 2003).

For example (Kimen 2003), if we want to develop a website, some of the basic deliverables must be include:

Site Maps: Maps reflect navigation and main content buckets. They are usually constructed to look like flowcharts and show how users navigate from one section to another.

Content Maps: Detailed maps that show what exists on each page and how content on some pages interacts with content on other pages.

Page Schematics: Black and white line drawings or block diagrams to hand off to a visual designer. These may, or may not, reflect layout and are used mostly to inform the designer and the client exactly what information, links, content, promotional space, and navigation will be on every page of the site. Schematics also help illustrate priority.

Text-Based Outlines: Sometimes information architects want to show architecture as indented text outlines and lists.

Interactive, Semi-Functional Prototyping: In some cases, information architects are responsible for outlining or storyboarding functional prototypes, and in others they actually build prototypes with HTML, Flash, Director, or PowerPoint.

2.4.4 How do architects evaluate or design an information system?

First, even before evaluating an existing information system for architectural

improvements, it's extremely important to find out who's using it, who's building it, and what its goals are. Maybe the hardest part of information architecture is to help identify a focus--a necessary component of intuitive function. But after focusing, evaluation is all about anticipated user paths, logical process flows, and determining how to balance efficiency with ease of use. Good, consistent information architecture will help users build relationships and trust with the technology and product. So, a good place to start is to look for the ways information system are, and are not, consistent (Kimen 2003).

When designing a new information system, it's always best to start with all the pieces, though this is seldom the case. You'll probably be hard-pressed to find a client who didn't change their minds half a dozen times over three phases of project architecture. And architects can change their minds because it is often difficult to predict all the pieces beforehand. It is the responsibility of the design firm and architect to ask the right questions, and it's the client's responsibility to understand what they are trying to build (Kimen 2003).

Architecture can, and should be, an extremely collaborative and iterative process, which evolves somewhat organically in as much structure that can be defined up-front as possible. Anything an IA can do to ask as many questions and get as many answers up-front will ultimately help the process. Architects also need to focus on who will be using the information system, strategic and business goals, key usability principals, technical constraints, and future needs (Kimen 2003).

2.4.5 Styles of information architecture

There are two main approaches to defining information architecture (Barker n.d.). These are:

Top-down information architecture

This involves developing a broad understanding of the business strategies and

user needs, before defining the high level structure of information system, and finally the detailed relationships between content.

Bottom-up information architecture

This involves understanding the detailed relationships between content, creating walkthroughs (or storyboards) to show how the system could support specific user requirements and then considering the higher level structure that will be required to support these requirements.

Both of these techniques are important in a project. A project that ignores top-down approaches may result in well-organized, findable content that does not meet the needs of users or the business. A project that ignores bottom-up approaches may result in an information system that allows people to find information but does not allow them the opportunity to explore related content (Barker n.d.).

2.4.6 How to create an effective information architecture?

An effective information architecture comes from understanding business objectives and constraints, the content, and the requirements of the people that will use the information system (Barker n.d.).

Information architecture is often described using Fig 2.4:

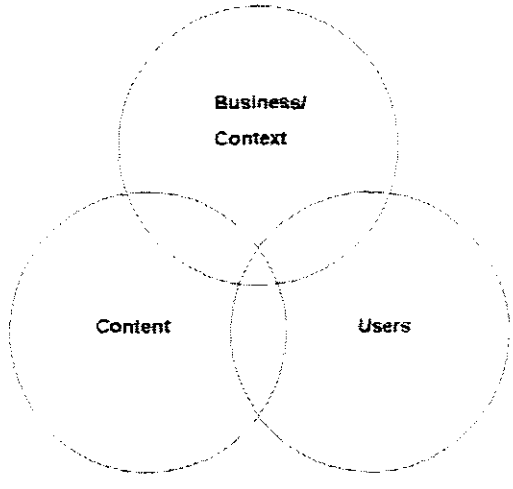


Fig 2.4 Information architecture (Barker n.d.)

Business/Context

Understanding an organizations' business objectives, politics, culture, technology, resources and constraints is essential before considering development of the information architecture.

Techniques for understanding context include:

1. *Reading existing documentation*

   Mission statements, organization charts, previous research and vision documents are a quick way of building up an understanding of the context in which the system must work.

2. Stakeholder interviews

   Speaking to stakeholders provides valuable insight into business context and can unearth previously unknown objectives and issues.

Content

The most effective method for understanding the quantity and quality of content (i.e. functionality and information) proposed for a system is to conduct a content inventory.

Content inventories identify all of the proposed content for a system, where the content currently resides, who owns it and any existing relationships between content.

Content inventories are also commonly used to aid the process of migrating content between the old and new systems.

2.4.7 Products from the information architecture process

Various methods are used to capture and define an information architecture (Barker n.d.). Some of the most common methods are:

1. System maps
2. Annotated page layouts
3. Content matrices
4. Page templates

There are also a number other possible by-products from the process (Barker n.d.). Such as:

1. Personas
2. Prototypes
3. Storyboards

Each of these methods and by-products is explained in detail below (Barker n.d.).

Site maps

System maps are perhaps the most widely known and understood deliverable from the process of defining an information architecture.

A system map is a high level diagram showing the hierarchy of a system. System maps reflect the information structure, but are not necessarily indicative of the navigation structure.

Annotated page layouts

Page layouts define page level navigation, content types and functional elements. Annotations are used to provide guidance for the visual designers and developers who

will use the page layouts to build the site.

Page layouts are alternatively known as wire frames, blue prints or screen details.

Content matrix

A content matrix lists each page in the system and identifies the content that will appear on that page.

Page templates

Page templates may be required when defining large-scale websites and intranets. Page templates define the layout of common page elements, such as global navigation, content and local navigation. Page templates are commonly used when developing content management systems.

Personas

Personas are a technique for defining archetypical users of the system. Personas are a cheap technique for evaluating the information architecture without conducting user research.

Prototypes can be used to bring an IA to life

Prototypes

Prototypes are models of the system. Prototypes can be as simple as paper-based sketches, or as complex as fully interactive systems. Research shows that paper-based prototypes are just as effective for identifying issues as fully interactive systems.

Prototypes are often developed to bring the information architecture to life, thus enabling users and other members of the project team to comment on the architecture before the system is built.

Storyboards

Storyboards are another technique for bringing the information architecture to life without building it. Storyboards are sketches showing how a user would interact with a system to complete a common task.

Storyboards enable other members of the project team to understand the proposed information architecture before the system is built.

2.4.8 Information architecture and usability

Some people find the relationship and distinction between information architecture and usability unclear (Barker n.d.).

Information architecture is not the same as usability, but the two are closely related. As described in a previous KM Column (Donna Maurer on www.steptwo.com.au), usability encompasses two related concepts:

1. Usability is an attribute of the quality of a system:

    "We need to create a usable intranet"

2. Usability is a process or set of techniques used during a design and development project:

    "We need to include usability activities in this project"

In both cases usability is a broader concept, whereas information architecture is far

more specific (Barker n.d.).

Information architecture as an attribute of the quality of a system

An effective information architecture is one of a number of attributes of a usable system. Other factors involving the usability of a system include:

1. Visual design
2. Interaction design
3. Functionality
4. Content writing

Information architecture as a process

The process for creating an effective information architecture is a sub-set of the usability activities involved in a project.

Although weighted to the beginning of the project, usability activities should continue throughout a project and evaluate issues beyond simply the information architecture (Barker n.d.).

2.4.9 Creating an effective information architecture in 9 steps

The following steps define a process for creating an effective information architecture (Barker n.d.).

1. Understand the business/contextual requirements and the proposed content for the system. Read all existing documentation, interview stakeholders and conduct a content inventory.

2. Conduct cards sorting exercises with a number of representative users.

3. Evaluate the output of the card sorting exercises. Look for trends in grouping and labelling.

4. Develop a draft information architecture (i.e. information groupings and hierarchy).

5. Evaluate the draft information architecture using the card-based classification evaluation technique. Don't expect to get the information architecture right the first time. Capturing the right terminology and hierarchy may take several iterations.

6. Document the information architecture in a map. This is not the final map, the map will only be finalized after layouts have been defined.

7. Define a number of common user tasks. This technique is known as storyboarding.

8. Walk other members of the project team through the storyboards and leave them in shared workspaces for comments.
If possible within the constraints of the project, it is good to conduct task-based usability tests on paper prototypes as it provides valuable feedback without going to the expense of creating higher quality designs.

9. Create detailed page layouts to support key user tasks. Layouts should be annotated with guidance for visual designers and developers.

2.4.10 Who creates the information architecture?

Increasingly, companies are realizing the importance of information architecture and are employing specialist 'information architects' to perform this role (Barker n.d.).

But information architecture is also defined by:

1. intranet designers and managers
2. website designers and managers
3. visual designers
4. other people designing information systems
5. programmers
6. librarians
7. technical writers

2.5 Conclusion

It simply isn't good enough for organizations to build functionality or write content, put it on their computer systems and expect people to be able to find it (Barker n.d.).

Developing an effective information architecture is an essential step in the development of all computer systems (Barker n.d.).

Effective information architectures enable people to quickly, easily and intuitively find content. This avoids frustration and increases the chance that the users will return to the system the next time they require similar information (Barker n.d.).

CHAPTER 3 METHODOLOGY

The purpose of this methodology is to provide the reader with general overview of Object Oriented Methodology and UML-based design methodology. This chapter focused on relevant methodology of object oriented design and UML-based design.

3.1 Object Oriented Methodology

3.1.1 OOM Overview (HKSAR 2004)

HKSAR (2004) *indicated: " Object Oriented Methodology (OOM) is a new system* development approach encouraging and facilitating re-use of software components. With this methodology, a computer system can be developed on a component basis which enables the effective re-use of existing components and facilitates the sharing of its components by other systems. By the adoption of OOM, higher productivity, lower maintenance cost and better quality can be achieved.

The ultimate objective of OOM is application assembly - the construction of new business solutions from existing components. The components are combined in different ways to meet the new requirements specified by the user community. Only completely new functionality will have to be built to complete the solution.

OOM applies a single object model that evolves from the analysis and design stage and carries all the way down to the programming level. An object contains both the data and the functions that operate upon that data. An object can only be accessed via the functions it makes publicly available, so that all details of its implementation are hidden from all other objects. This strong encapsulation provides the basis for the improvements in traceability, quality, maintainability and extensibility that are key features of well-designed Object Oriented systems.

The OOM life cycle consists of six stages:

1. Business planning;
2. Business architecture definition;
3. Technical architecture definition;
4. Incremental delivery planning;
5. Incremental design and build;
6. Deployment.

Business Planning

The Business Planning stage is usually conducted through a series of meetings and workshops with the business management and business users. These meetings initiate the development process by establishing a mutual understanding of the objectives, scope, user requirements and assess the feasibility of the development project.

The tasks of the Business Planning are:

1. Examine Current Environment and Define Objectives
2. Identify Overall Requirements
3. Model Business Processes
4. Analyze Use Cases
5. Model Initial Business Classes
6. Identify Reusable Components
7. Finalize Feasibility Study

Business Architecture Definition

Business Architecture Definition focuses on gaining an increased understanding of the users' needs and defining a solution that will satisfy the needs.

System requirements as specified in the Use Cases will be analysed. The component services required to fulfil the system functionality will be identified. Component services will be realised by designing the detailed business classes and their collaboration within the components.

In order to gain advantage from component based development, services available from reusable components and the dependencies between them will be identified. Parallel to the above activities, prototypes of screens and reports will be made.

The tasks of the Business Architecture Definition are:

1. Review Business Planning Deliverables
2. Prototype User Interface
3. Identify Candidate Components
4. Model Component Interaction
5. Model Business Class Interaction
6. Map Business Classes to Entities
7. Finalize Interaction Model

Technical Architecture Definition

The activities of the Technical Architecture Definition (TAD) will be executed in parallel to the Business Architecture Definition (BAD) stage. While BAD focus on the business requirements, the TAD stage focuses on technology and architecture sides.

The tasks of the Technical Architecture Definition are:

1. Identify Technical Requirements
2. Select Industry Standard Solutions
3. Describe Technical System Architecture
4. Define Technical Application Architecture

5. Design Technical Services and Patterns
6. Prototype Technical Architecture
7. Finalize Technical Architecture

Incremental Delivery Planning

An increment delivers a number of complete Use Cases. Since each Use Case describes a way in which the system can be used, they can deliver value from the moment of deployment. Based on the prioritisation of Use Cases developed in the Business Planning stage, the most important Use Cases are delivered first.

The tasks of the Incremental Delivery Planning are:

1. Define Increments and Implementation Strategy
2. Finalize Systems Analysis and Design

Incremental Design and Build

This stage will take the final deliverables of the Business Architecture Definition as its starting point. The components and their classes will be mapped onto the technical architecture from the Technical Architecture Definition and will subsequently be optimised for reasons of performance, maintainability etc.

The optimised classes will be realised using the class concept, provided as part of the programming language. The programs will be tested, accepted by users and made ready for deployment.

The tasks of the Incremental Design and Build Business are:

1. Consolidate the Architectures
2. Design and Build Solution Process
3. Design and Build Business Components

4. Design and Build Data Services
5. Test Increment

Deployment

Incremental deployment could be adopted depending on the nature of the business, project constraints, and the benefits achievable.

The tasks of the Deployment are:

1. Prepare User Training
2. Conduct User Training
3. Perform Data Conversion
4. Prepare for Delivery to Production Environment
5. Deliver to Production Environment"

3.1.2 Object-oriented design theory

Object-oriented software is all about objects. An object is a "black box" which receives and sends messages. A black box actually contains code (sequences of computer instructions) and data (information which the instructions operate on). Traditionally, code and data have been kept apart. For example, in the C language, units of code are called functions, while units of data are called structures. Functions and structures are not formally connected in C. A C function can operate on more than one type of structure, and more than one function can operate on the same structure (Montlick 1999).

Not so for object-oriented software! In o-o (object-oriented) programming, code and data are merged into a single indivisible thing -- an object. This has some big advantages. But first, here is why we developed the "black box" metaphor for an object. A primary rule of object-oriented programming is this: as the user of an object,

you should never need to peek inside the box (Montlick 1999)!

Why shouldn't you need to look inside an object? For one thing, all communication to it is done via messages. The object which a message is sent to is called the receiver of the message. Messages define the interface to the object. Everything an object can do is represented by its message interface. So you shouldn't have to know anything about what is in the black box in order to use it (Montlick 1999).

And not looking inside the object's black box doesn't tempt you to directly modify that object. If you did, you would be tampering with the details of how the object works. Suppose the person who programmed the object in the first place decided later on to change some of these details? Then you would be in trouble. Your software would no longer work correctly! But so long as you just deal with objects as black boxes via their messages, the software is guaranteed to work. Providing access to an object only through its messages, while keeping the details private is called information hiding. An equivalent buzzword is encapsulation (Montlick 1999).

Why all this concern for being able to change software? Because experience has taught us that software changes. A popular adage is that "software is not written, it is re-written". And some of the costliest mistakes in computer history have come from software that breaks when someone tries to change it (Montlick 1999).

Object

There are many definitions of an object, such as found in (Booch 1991): "An object has state, behaviour, and identity; the structure and behaviour of similar objects are defined in their common class; the terms instance and object are interchangeable". This is a "classical languages" definition, as defined in (Coplien 1992), where "classes play a central role in the object model", since they do not in prototyping/delegation

languages. "The term object was first formally applied in the Simulate language, and objects typically existed in Simulate programs to simulate some aspect of reality" (Booch 1991). Other definitions referenced by Booch include Smith and Tockey: "an object represents an individual, identifiable item, unit, or entity, either real or abstract, with a well-defined role in the problem domain." and (Cox 1991): "anything with a crisply defined boundary" (in context, this is "outside the computer domain". A more conventional definition appears on pg 54). Booch goes on to describe these definitions in depth. (Martin & Odell 1992) defines: "An "object" is anything to which a concept applies", and "A concept is an idea or notion we share that applies to certain objects in our awareness". (Rumbaugh 1991) defines: "We define an object as a concept, abstraction or thing with crisp boundaries and meaning for the problem at hand." (Shlaer & Mellor 1988) defines: "An object is an abstraction of a set of real-world things such that:

1. All of the real-world things in the set - the instances - have the same characteristics
2. All instances are subject to, and conform to the same rules"

And on identifying objects: "What are the *things* in this problem? Most of the things are likely to fall into the following five categories: Tangible things, Roles, Incidents, Interactions, and Specifications." (Booch 1991) covers "Identifying Key Abstractions" for objects and classes based on an understanding of the problem domain and (Jacobson 1992) provides a novel approach of identifying objects through use-cases (scenarios), leading to a use-case driven design.

The implementation of objects could roughly be categorized into descriptor-based, capability-based, and simple static-based approaches. Descriptor-based approaches (e.g. Smalltalk handles) allow powerful dynamic typing, as do the capability-based approaches which are typically found in object-oriented databases and operating systems (object id's). A "proxy" based approach with an added layer of indirection to Smalltalk's handles is found in Distributed Smalltalk which allows transparent,

distributed, and migrating objects (Kim & Lochovsky 1989 and Gao & Yuen 1993). Simple static approaches are found in languages such as C++, although the new RTTI facility will supply simple dynamic typing (checked downcasting) similar to those introduced by Eiffel in 1989 through the notion of assignment attempt, also known as *type narrowing*.

Descriptor-based approaches can have pointer semantics and can be statically typeless (or just "typeless", as in Smalltalk) where references (variables) have no type, but the objects (values) they point to always do. An untyped pointer (such as void* in C++) and an embedded dynamic typing scheme are used in more conventional languages to fully emulate this style of dynamically typed programming (Hathaway 1995).

A virtual member in statically typed languages is a base class member that can be set or respecified by a derived class. This is roughly equivalent to a pointer or function pointer in the base class being set by the derived class. (Stroustrup 1990) covers the implementation details of virtual member functions in C++, which also involve an offset for the receiver to handle multiple-inheritance. This is an example of dynamic binding, which replaces a switch statement on variant parts with a single call, reducing code size and program complexity (fewer nested programming constructs) and allowing variants to be added without modifying client code (which causes higher defect injection rates during maintenance and debugging) (Hathaway 1995).

Virtual members in dynamically typed languages are more flexible because static type checking requirements are dropped (Hathaway 1995).
The terms method/member function, instance variable/member data, subclass/derived class, parent class/base class, etc. will be used interchangeably. As pointed out in (Stroustrup 1990), the base/derived class terminology may be preferable to the sub/super-class terminology, and is preferred in this document also.

Delegation/prototyping languages (Kim & Lochovsky 1989; Ungar & Smith 1987 &

Sciore 1989) have a more flexible kind of object which can play the role of classes in classical OO languages. Since there is no separate class construct in these languages, and only objects, they are referred to as single-hierarchy, or 1 Level systems. Objects contain fields, methods and delegates (pseudo parents), whereas classical object-oriented languages associate method, field and parent definitions with classes (and only associate state and class with objects, although tables of function pointers for dynamic binding is an exception). However, one-level objects often play the role of classes to take advantage of sharing and often instances will simply delegate to parents to access methods or shared state, otherwise idiosyncratic objects, a powerful and natural concept, will result. Typical 1 Level objects can contain any number of fields, methods and parents and any object can be used as a template/exemplar, thus performing the classical role of a class. In typical prototyping systems, parents (as any other member) can be added or changed dynamically, providing dynamic multiple inheritance (or more typically simple delegation). Here, the term "Prototype" usually refers to prototype theory, a recent theory of classification where any object can be inherited from or cloned to serve as a prototype for newly created instances. (The Author also uses the term for languages providing high quality support for rapid prototyping, although this usage is atypical) See (Booch 1994) for a brief discussion of prototype theory in the context of OOA and OOD.

It is common in such systems for an object to "become" another kind of object by changing its parent. A good example is a window becoming an icon, as window and icon objects display different behaviour (although cognitive differences are significant too. Delegation refers to delegating the search for an attribute to a delegate, and is therefore more of a pure message passing mechanism (as with dynamic scoping) than inheritance, which also typically specifies non-shared state when used for representation (Hathaway 1995).

Chambers has proposed an interesting variation called "Predicate Classes" (Chambers 1993) as a part of his Cecil language. These classes will only be parents when

certain predicates are true. This can support a types/classes as collections of objects view, which is the same as the types as sets of values view taken by (Cardelli & Wegner 1985). (Martin & Odell 1992) provides some examples of this view applied during OOA.

Object Encapsulation (Or Protection)

(Booch 1991) defines: "Encapsulation is the process of hiding all of the details of an object that do not contribute to its essential characteristics."

(Coad & Yourdon 1991) defines: "Encapsulation (Information Hiding). A principle, used when developing an overall program structure, that each component of a program should encapsulate or hide a single design decision... The interface to each module is defined in such a way as to reveal as little as possible about its inner workings. (Oxford, 1986)"

Some languages permit arbitrary access to objects and allow methods to be defined outside of a class as in conventional programming. Simulate and Object Pascal provide no protection for objects, meaning instance variables may be accessed wherever visible. CLOS and Ada allow methods to be defined outside of a class, providing functions and procedures. While both CLOS and Ada have packages for encapsulation, CLOS's are optional while Ada's methodology clearly specifies class-like encapsulation (Adts) (Hathaway 1995).

However, most object-oriented languages provide a well defined interface to their objects thru classes. C++ has a very general encapsulation/protection mechanism with public, private and protected members. Public members (member data and member functions) may be accessed from anywhere. A Stack's Push and Pop methods will be public. Private members are only accessible from within a class. A Stack's representation, such as a list or array, will usually be private. Protected

members are accessible from within a class and also from within subclasses (also called derived classes). A Stack's representation could be declared protected allowing subclass access. C++ also allows a class to specify friends (other (sub)classes and functions), that can access all members (its representation). Eiffel 3.0 allows exporting access to specific classes (Hathaway 1995).

For another example, Smalltalk's class instance variables are not accessible from outside of their class (they are not only private, but invisible). Smalltalk's methods are all public (can be invoked from anywhere), but a private specifier indicates methods should not be used from outside of the class. All Smalltalk instance variables can be accessed by subclasses, helping with abstract classes and overriding (Hathaway 1995).

Another issue is per-object or per-class protection. Per-class protection is most common (e.g. Ada, C++, Eiffel), where class methods can access any object of that class and not just the receiver. Methods can only access the receiver in per-object protection. This supports a sub typing model, as any object other than the receiver is only satisfying an abstract type interface, whereby no method or object structure can be inferred in the general case (Hathaway 1995).

Class

A class is a general term denoting classification and also has a new meaning in object-oriented methods. Within the OO context, a class is a specification of structure (instance variables), behaviour (methods), and inheritance (parents, or recursive structure and behaviour) for objects. As pointed out above, classes can also specify access permissions for clients and derived classes, visibility and member lookup resolution. This is a feature-based or intensional definition, emphasizing a class as a descriptor/constructor of objects (as opposed to a collection of objects, as with the more classical extensional view, which may begin the analysis process)

(Hathaway 1995).

Original Aristotlean classification defines a "class" as a generalization of objects: (Booch 1991) "A group, set, or kind marked by common attributes or a common attribute; a group division, distinction, or rating based on quality, degree of competence, or condition".
(Booch's definition in the context of OOD) "A class is a set of objects that share a common structure and a common behaviour." "A single object is simply an instance of a class."

The intension of a class is its semantics and its extension is its instances (Martin & Odell 1992).

(Booch 1994) proposes 3 views of classification as useful in OO analysis and design: classical categorization (common properties), conceptual clustering (conceptual descriptions), and prototype theory (resemblance to an exemplar). He advocates starting with the former approach, turning to the second approach upon unsatisfactory results, and finally the latter if the first two approaches fail to suffice.

Meta-Class

A Meta-Class is a class' class. If a class is an object, then that object must have a class (in classical OO anyway). Compilers provide an easy way to picture Meta-Classes. Classes must be implemented in some way; perhaps with dictionaries for methods, instances, and parents and methods to perform all the work of being a class. This can be declared in a class named "Meta-Class". The Meta-Class can also provide services to application programs, such as returning a set of all methods, instances or parents for review (or even modification). (Booch 1991) provides another example in Smalltalk with timers. In Smalltalk, the situation is more complex. To make this easy, refer to the following listing, which is based on the

number of levels of distinct instantiations:

1 Level System

All objects can be viewed as classes and all classes can be viewed as objects (as in Self). There is no need for Meta-Classes because objects describe themselves. Also called "single-hierarchy" systems. There is only 1 kind of object.

2 Level System

All Objects are instances of a Class but Classes are not accessible to programs (no Meta-Class except for in the compiler and perhaps for type-safe linkage, as in C++). There are 2 kinds of distinct objects: objects and classes.

3 Level System

All objects are instances of a class and all classes are instances of Meta-Class. The Meta-Class is a class and is therefore an instance of itself (really making this a 3 1/2 Level System). This allows classes to be first class objects and therefore classes are available to programs. There are 2 kinds of distinct objects (objects and classes), with a distinguished class, the meta-class.

5 Level System

What Smalltalk provides. Like a 3 Level System, but there is an extra level of specialized Meta-Classes for classes. There is still a Meta-Class as in a 3 Level System, but as a class it also has a specialized Meta-Class, the "Meta-Class class" and this results in a 5 Level System:

object

class

class class (Smalltalk's Meta-Classes)

Meta-Class

Meta-Class class

The "class class" handle messages to classes, such as constructors and "new", and also "class variables" (a term from Smalltalk), which are variables shared between all

instances of a class (static member data in C++). There are 3 distinct kinds of objects (objects, classes, and meta-classes) (Hathaway 1995).

Inheritance

Inheritance provides a natural classification for kinds of objects and allows for the commonality of objects to be explicitly taken advantage of in modelling and constructing object systems. Natural means we use concepts, classification, and generalization to understand and deal with the complexities of the real world. See the example below using computers (Hathaway 1995).

Inheritance is a relationship between classes where one class is the parent (base/superclass/ancestor/etc.) class of another. Inheritance provides programming by extension (as opposed to programming by reinvention (LaLonde & Pugh 1990)) and can be used as an is-a-kind-of (or is-a) relationship or for differential programming. Inheritance can also double for assignment compatibility.

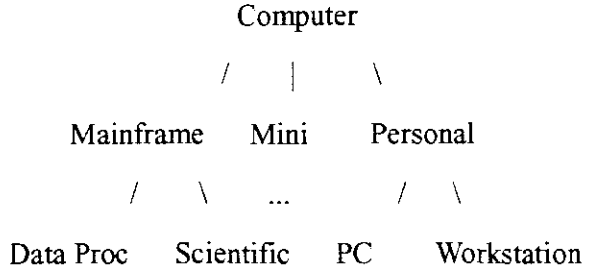
In delegation languages, such as Self, inheritance is delegation where objects refer to other objects to respond to messages (environment) and do not respecify state by default (Hathaway 1995).

Inherited parents can specify various flavours of state. Delegation languages don't specify new state by default (to do so requires cloning), C-based (C++, Objective-C, etc.), lisp-based (CLOS, Flavours, Scheme, etc.), and Pascal-based (Ada95, Modula-3, Object Pascal, etc.) OO languages do, but with multiple-inheritance can also share parents within a class lattice (CLOS and Eiffel provide this as a default at the level of slots and features, respectively).

Defining inheritance (with a thorough description or denotational semantic definition, or both) can avoid confusion about which inheritance scheme is being used

(especially in OOD), because inheritance has many variations and combinations of state and environment (sometimes with complex rules). Inheritance can also be used for typing, where a type or class can be used to specify required attributes of a matching object (Cardelli & Wegner 1985)). It would be more judicious to have discussions on how inheritance should be defined instead of over what it is, since it has many existing uses and semantics.

An example of the is-a-kind-of relationship is shown below. Is-a is often used synonymously, but can be used to show the "object is-a class" instantiation relationship. In classical OO, inheritance is a relationship between classes only. In one-level systems, is-a (object instantiation) and is-a-kind-of (inheritance) are merged into one (Ungar & Smith 1987, Madsen, Pedersen & Nygaard 1993, Sciore 1989).



Class hierarchies are subjective (Booch 1991) and (Lakoff 1987) and usually drawn with the parent class on top, but more demanding graphs (as is often the case in (Rumbaugh 1991)) allow any topology, with the head of an arrow indicating the base class and the tail indicating the derived class.

Differential programming is the use of inheritance to reuse existing classes by making a small change to a class. Creating a subclass to alter a method or to add a method to a parent class is an example (Hathaway 1995).

Multiple Inheritance

Multiple Inheritance occurs when a class inherits from more than one parent. For example, a person is a mammal and an intellectual_entity, and a document may be an editable_item and a kind of literature (Hathaway 1995).

Mixin's is a style of MI (from flavours) where a class is created to provide additional attributes or properties to other classes. They are intended to be inherited by any class requiring them. Method combination, or calling sequences of before, after, and around methods or even several primary methods (Kim & Lochovsky 1989), make good use of mixins by invoking their methods without explicitly calling them, allowing client class code to remain unchanged (Booch 1991).

Dynamic Inheritance

Dynamic inheritance allows objects to change and evolve over time. Since base classes provide properties and attributes for objects, changing base classes changes the properties and attributes of a class. A previous example was a window changing into an icon and then back again, which involves changing a base class between a window and icon class (Hathaway 1995).

More specifically, dynamic inheritance refers to the ability to add, delete, or change parents from objects (or classes) at run-time. Actors, CLOS, and Smalltalk provide dynamic inheritance in some form or other. Single hierarchy systems, such as Self, provide dynamic inheritance in the form of delegation (Ungar & Smith 1987).

See also (Kim & Lochovsky 1989) for a discussion and (Coplien 1992) for some implementation discussion in C++.

Shared (Repeated) Inheritance

Multiple Inheritance brings up the possibility for a class to appear as a parent more

than once in a class graph (repeated inheritance), and there is then a potential to share that class. Only one instance of the class will then appear in the graph (as is always the case in CLOS, because all *members* with the same name will be shared (receive a single slot) with the greatest common subtype as its type). C++ provides an alternative, where only parents specified as virtual (virtual bases) are shared within the same class lattice, allowing both shared and non-shared occurrences of a parent to coexist. All "features" in Eiffel (C++ members) of a repeated parent that are not to be shared must be renamed "along an inheritance path", else they are shared by default. This allows a finer granularity of control and consistent name resolution but requires more work for parents with many features (Hathaway 1995).

Specialization/Generalization/Overriding

To create a subclass is specialization, to factor out common parts of derived classes into a common base (or parent) is generalization (Booch 1991). Overriding is the term used in Smalltalk and C++ for redefining a (virtual in Simula and C++) method in a derived class, thus providing specialized behaviour. All routines in Smalltalk are overridable and non-"frozen" features in Eiffel can be "redefined" in a derived class. Whenever a method is invoked on an object of the base class, the derived class method is executed overriding the base class method, if any. Overriding in Simula is a combination of overloading and multiple-polymorphism because parameters do not have to be declared. Eiffel and BETA are examples of languages allowing any member to be redefined and not just methods, as is typical (Hathaway 1995).

Method (And Receiver and Message)

A method implements behaviour, which is defined by (Booch 1991):
Behaviour is how an object acts and reacts, in terms of its state changes and message passing.

A method is a function or procedure which is defined in a class and typically can access the internal state of an object of that class to perform some operation. It can be thought of as a procedure with the first parameter as the object to work on. This object is called the receiver, which is the object the method operates on. An exception exists with C++'s static member functions which do not have a receiver, or "this" pointer. The following are some common notations for invoking a method, and this invocation can be called a message (or message passing, see below) (Hathaway 1995):

receiver.message_name(a1, a2, a3)

receiver message_name: a1 parm1: a2 parm3: a3

Selector would be another good choice for message_name in the above examples, although keywords (or formal parameter names, like named parameters) are considered part of the selector in Smalltalk (and hence Objective-C).

If done statically, this can be referred to as invocation, and message passing if done dynamically (true dynamic binding). Statically typed dynamic binding (e.g. C++ and Eiffel) is really in between (checked function pointers) (Hathaway 1995).

Multi-Methods And Multiple-Polymorphism

Multi-methods involve two primary concepts, multiple-polymorphism and lack of encapsulation. These issues are orthogonal. Multiple-polymorphism implies more than one parameter can be used in the selection of a method. Lack of encapsulation implies all arguments can be accessed by a multi-method (although packages can be used to restrict access, as in CLOS). Multi-methods can also imply a functional prefix notation, although the CLOS designers (who coined the term "multi-method") consider the functional and receiver based forms (messages) equivalent. Functional syntax was chosen "in order to minimize the number of new mechanisms added to

COMMON LISP" (Kim & Lochovsky 1989). (Chambers 1993) discusses multi-methods in his new OO language, Cecil.

Multiple-polymorphism allows specialized functions or methods to be defined to handle various cases:

```
+(int, int)
+(int, float)
+(int, complex)
+(int, real)
+(float, complex)
+(float, real)
+(float, float)
```

The above functions are specialized to each of the cases required allowing single, highly cohesive and loosely coupled functions to be defined. This is also the true essence of object-oriented polymorphism, which allows objects to define methods for each specific case desired. In addition to better coupling and cohesion, multiple-polymorphism reduces program complexity by avoiding coding logic (switch statements) and because small methods further reduce complexity, as code complexity doesn't grow linearly with lines of code per method, but perhaps exponentially. This should be distinguished from double dispatch, a fancy name for single dispatch after a call, which only provides switching on a single argument per call (but for 2 levels), consistently ignoring the inherent type of parameters in messaging. Double dispatch is used in languages with static typing for simplicity and efficiency considerations (Hathaway 1995).

If all of the above types are Numbers, code can be written without concern for the actual classes of objects present:

```
fn(one, two: Number): Number
    return one + two;
```

The addition expression above will invoke the correct "+" function based on the inherent (true, actual, or dynamic) types of one and two. Only the inherent type of "one" would be used with double dispatch! Further, double dispatch would only allow switching to the "fn" function based on the type of "one" also. This could lead to the use of switch statements based on type or complex coding in many real-world programming situations, unnecessarily. This should only be used as necessary, e.g. if the implementation language doesn't support multiple-polymorphism and either efficiency considerations dominate and double dispatch can be suffered, or an embedded dynamic typing scheme is used (Hathaway 1995).

Why do multi-methods allow open access to parameters? It allows efficient handling, like C++ friends, usually by allowing representation details of more than one object to be exposed. See (Kim & Lochovsky 1989) for an alternative explanation. While open access can be useful in some cases, it typically isn't recommended as a general OO practice and also violates subtype polymorphism, because only subclass polymorphism is based on representation and not type.

Polymorphic languages can be statically typed to provide strong type checking, efficiency, and to support a static programming idiom, but require restrictions in many cases, such as requiring overriding methods to have identical signatures with the methods they substitute (as in C++) or allowing covariant parameters but limiting base class usage (as in Eiffel). If these restrictions are dropped, multiple-polymorphism results. Thus a single overridable function declared in a base class may have several functions overriding it in a derived class differentiated only by their formal argument types. This therefore requires both static and dynamic typing, because no formal argument differentiation is possible without static types, as in Smalltalk, and no actual argument differentiation is possible without dynamic types (as in C++ and Eiffel) (Hathaway 1995).

Polymorphism

Polymorphism is a ubiquitous concept in object-oriented programming and is defined in many ways, so many definitions are presented from: Websters', Author, Strachey, Cardelli and Wegner, Booch, Meyer, Stroustrup, and Rumbaugh. Polymorphism is often considered the most powerful facility of an OOPL.

Webster's New World Dictionary:

1.State or condition of being polymorphous.

2.Cryall. crystallization into 2 or more chemically identical but crystallographically distinct forms.

3.Zool., Bot. existence of an animal or plant in several forms or color varieties.

Author's Definition:

Polymorphism is the ability of an object (or reference) to assume (be replaced by) or become many different forms of object. Inheritance (or delegation) specifies slightly different or additional structure or behavior for an object, and these more specific or additional attributes of an object of a base class (or type) when assuming or becoming an object of a derived class characterizes object-oriented polymorphism. This is a special case of parametric polymorphism, which allows an object (or reference) to assume or become any object (possibly satisfying some implicit or explicit type constraints (parametric type), or a common structure), with this common structure being provided by base classes or types (subclass and subtype polymorphism, respectively).

"Poly" means "many" and "morph" means "form". The homograph polymorphism has many uses in the sciences, all referring to objects that can take on or assume many different forms. Computer Science refers to Strachey's original definitions of polymorphism, as divided into two major forms, parametric and ad-hoc. Cardelli and Wegner follow up with another classification scheme, adding inclusion polymorphism for subtyping and inheritance.

Dynamic Binding

Dynamic binding has two forms, static and dynamic. Statically-typed dynamic binding is found in languages such as C++ (virtual functions) and Eiffel (redefinition). It is not known which function will be called for a virtual function at run-time because a derived class may override the function, in which case the overriding function must be called. Statically determining all possibilities of usage is undecidable. When the complete program is compiled, all such functions are resolved (statically) for actual objects. Formal object usage must have a consistent way of accessing these functions, as achieved thru vtables of function pointers in the actual objects (C++) or equivalent, providing statically-typed dynamic binding (this is really just defining simple function pointers with static type checking in the base class, and filling them in the derived class, along with offsets to reset the receiver) (Hathaway 1995).

The run-time selection of methods is another case of dynamic binding, meaning lookup is performed (bound) at run-time (dynamically). This is often desired and even required in many applications including databases, distributed programming and user interaction (e.g. GUIs). Examples can be found in Simson , Garfinkel & Michael (1993) and Cox (1991). To extend Garfinkel's example with multiple-polymorphism, a cut operation in an Edit submenu may pass the cut operation (along with parameters) to any object on the desktop, each of which handles the message in its own way (OO). If an (application) object can cut many kinds of objects such as text and graphical objects, multiple-polymorphism comes into play, as many overloaded cut methods, one per type of object to be cut, are available in the receiving object, the particular method being selected based on the actual type of object being cut (which in the GUI case is not available until run-time) (Hathaway 1995).

Again, various optimizations exist for dynamic lookup to increase efficiency (such as

found in (Agrawal 91) and (Chambers 1992)).

Dynamic binding allows new objects and code to be interfaced with or added to a system without affecting existing code and eliminates switch statements. This removes the spread of knowledge of specific classes throughout a system, as each object knows what operation to support. It also allows a reduction in program complexity by replacing a nested construct (switch statement) with a simple call. It also allows small packages of behaviour, improving coherence and loose coupling. Another benefit is that code complexity increases not linearly but exponentially with lines of code, so that packaging code into methods reduces program complexity considerably, even further that removing the nested switch statement! (Martin & Odell 1992) covers some of these issues.

A Separation Between Type And Class (Representation)

Subtype Polymorphism, as opposed to Subclass Polymorphism, is the best answer in OO. Parametric polymorphism is a related concept where this is also true, but is of a different flavour (and usually requires object attributes by use) (Hathaway 1995).

A type can be considered a set of values and a set of operations on those values. This can insure type-safe programming. However, the representation of types (classes in OO) can be separated from the notion of type allowing many representations per type while still maintaining reasonable type-safety (Hathaway 1995).

In many languages, a type has a single representation insuring all operations performed on that type are well defined (statically bound) and providing for efficiency by taking advantage of that representation wherever used. In many OO languages, sub classing and dynamic binding provides for greater flexibility by providing object specialization. However, in many OO languages classes are used for assignment

compatibility forcing an assigned object to inherit (transitively) from any polymorphic object's class (inclusion polymorphism based on class, or subclass polymorphism). This insures all operations to be performed on any polymorphic object are satisfied by any replacing objects. This also insures all types share a common representation, or at least a common base interface specification (Hathaway 1995).

By separating type from class, or representation (or perhaps separating class from type, by the aforementioned definition of type), a replacing object must satisfy the operations or type constraints of a polymorphic object (subtype polymorphism) but are not required to do so by an inheritance relation (subclass polymorphism), as is typical in most OOPLs. Dropping this restriction is somewhat less type-safe, because accidental matches of method signatures can occur, calling for greater care in use. (Black 1986) discusses this issue in Emerald. The same issue arises in parametric polymorphism (generics/templates), as any method matching a required signature is accepted, calling for careful matching of actual and formal generic parameters. The difference between static and dynamic binding in OO and dynamic binding and sub typing seems similar. A possible loss of semantic integrity/similarity is contrasted with greater power (Hathaway 1995).

It is possible to specify desired abstract properties of type specifications with mechanisms similar to Eiffel's pre-, post-, and invariant conditions. This helps to *insure the semantic integrity of replacing objects and their behaviour.* (Liskov & Wing 1993) provides a recent exposition.

Abstract classes ((Stroustrup 1991) and (Meyer 1988)) in typing provide a facility similar to subtype polymorphism; however, ACs require type compatible classes to inherit from them, providing a subclass polymorphism facility, and ACs can also specify representation. Sub typing is therefore most useful to avoid spreading knowledge of classes throughout a system, which is a high priority for loosely coupled modules and in distributed programming (Black et al. 1987).

The formal type system found in (Cardelli & Wegner 1985), Emerald/Jade (Black 1986) and (Raj & Levy 1989), original trellis/Owl, an experimental C++ extension (See Appendix E, Signatures), Sather (originally Eiffel-based), and an Eiffel superset (Jones 1992) are all examples of OO systems providing subtype polymorphism. Functional languages such as ML, Russell, and Haskell provide a separation with pure parametric polymorphism (as is also commonly found in OO languages in addition to inclusion polymorphism).

Generics and Templates

Generics (or Templates in C++) refer to the ability to parameterize types and functions with types. This is useful for parameterized classes and polymorphic functions as found in languages such as Ada, C++, Eiffel, and etc., although these are "syntactic" or restricted forms (Cardelli & Wegner 1985). Generics are orthogonal to inheritance, since types (and classes) may be generically parameterized. Generics provide for reusability in programming languages. An example is a Stack with a generically parameterized base type. This allows a single Stack class to provide many instantiations such as a Stack of ints, a Stack of any fundamental or user defined type, or even a Stack of Stacks of... Another example is a polymorphic sort function taking a base type with a comparison operator. The function can be called with any type (containing a comparison operator). See (Booch 1987) for several examples in Ada and (Murray 1993) for examples in C++.

While generics have many advantages, typical limitations include a static nature, which is an advantage for strong type checking but a potential disadvantage when causing dynamic compilation (leading to a time/space efficiency tradeoff), and sources can cause inlining and create source code dependencies and expand code size (unlike a single-body or "true" parametrically polymorphic implementation. Generics can also be viewed as a special case of type variables (Hathaway 1995).

Functions are typically generic in statically-typed parametrically-polymorphic languages. One such popular functional language is ML, in which all functions are generic. Russell and Haskel are more modern variants (Hathaway 1995).

3.2 A UML-Based Design Methodology

In software engineering, Unified Modelling Language (UML) is a non-proprietary, third generation modelling and specification language. However, the use of UML is not restricted to model software. It can be used for modelling hardware (engineering systems) and is commonly used for business process modelling, organizational structure, and systems engineering modelling. The UML is an open method used to specify, visualize, construct, and document the artefacts of an object-oriented software-intensive system under development. The UML represents a compilation of best engineering practices which have proven to be successful in modelling large, complex systems, especially at the architectural level (Answers.com).

UML succeeds the concepts of Booch, OMT and OOSE by fusing them into a single, common and widely usable modelling language. UML aims to be a standard modelling language which can model concurrent and distributed systems (OMG 2002).

UML is not an industry standard, but is taking shape under the auspices of the Object Management Group (OMG). OMG has called for information on object-oriented methodologies, that might create a rigorous software modelling language. Many industry leaders have responded in earnest to help create the standard (Biography.ms).

3.2.1 The primary artefacts of the Unified Modelling Language (Rational Software)

This question can be answered in two ways: first in terms of the artefacts that constitute the Unified Modelling Language (the inside view) and second in terms of the artefacts that users apply to model systems using the Unified Modelling Language (the outside view).

From the inside, the Unified Modelling Language consists of three things:

1. A formal meta-model
2. A graphical notation
3. A set of idioms of usage

The purpose of the meta-model was to provide a single, common, and unambiguous statement of the syntax and semantics of the elements of the Unified Modelling Language. The presence of this meta-model has made it possible for us to agree on semantics, decoupled from the human factors issues of how those semantics would best be rendered. Additionally, the meta-model has made it possible for us to explore ways to make the modelling language much simpler by, in a sense, unifying the elements of the Unified Modelling Language. The graphical notation is the most visible part of the Unified Modelling Language, and it constitutes the graphical syntax that humans and tools use to model systems. Lastly, the Unified Modelling Language encompasses a set of idioms that describe common usage (by humans) and degrees of freedom (by tools).

From the outside, the Unified Modelling Language encompasses a number of models that can be rendered in a variety of projections:

Use-case diagrams

Class diagrams

State-machine diagrams

Message-trace diagrams

Object-message diagrams

Process diagrams

Module diagrams

Platform diagrams

Use-case diagrams are as found in Objectory, and they serve to organize the use cases that encompass a system's behaviour. Class diagrams, derived from Booch and OMT, capture the static semantics of the classes that constitute a system. State-machine diagrams, derived from David Harel, capture the dynamic semantics of a class. Message-trace diagrams, object-message diagrams, and process diagrams, derived from Booch, OMT, and Fusion, capture the dynamic semantics of collaborations of objects. Module diagrams serve to model the development view of a system, whereas platform diagrams serve to model the physical computing topology upon which a system executes.

These are the primary artefacts that a modeller sees, although the method (and tools) provide for a number of derivative views.

3.2.2 The Unified Modelling Language meta-model (Rational Software)

A meta-model is a model of a model.

Think of it this way: Developers create things like class diagrams and object diagrams to model a system under development; this resulting model describes a system. A meta-model is a model of that model.

Meta-models are important, because they can provide a single, common, and unambiguous statement of the syntax and semantics of a model. We began our work on the Unified Modelling Language by starting with a meta-model, because it let us come to rapid agreement about the meaning of things (and the very things themselves) that were to constitute our unification. Thus, our meta-model includes some obvious

things like classes and objects (which are isomorphic to the elements of the modelling language) and some nonobvious things like uninterpreted and nonclass decls (which are artefacts of the meta-model).

For most users, the meta-model is invisible (as it should be). It's valuable to us, because it lets us communicate our intended semantics to each other and to tool builders. It also gives us something to throw stones at as we try new modelling problems: If we can model something complex easily, then we know we are on the right path. Similarly, it gives us something to evolve. A large part of our latest work has been making the meta-model simpler (which is a very hard thing to do).

Currently, the meta-model for the Unified Modelling Language is described in a combination of English text and class diagrams using the Unified Modelling Language itself. For those purists out there: Yes, Godel's Theorem does apply, which means that there are theoretical limits to what we can express about the meta-model using the meta-model itself. However, we think combination of text and diagrams strikes the right balance between expressiveness and readability.

Currently, we are working to make this model more formal, in two ways. First, we are writing a number of use cases against the meta-model, which lets us try out common and not-so-common modelling problems. Second, we are writing a number of use cases against the dynamic semantics of the meta-model, using formal logic. This is definitely not for the meek, but again it helps us be sure we are generating models that are self-consistent and precise.

3.2.3 The notation for the Unified Modelling Language (Rational Software)

The Unified Modelling Language notation is truly a melding of the graphical syntax of Booch, Objectory, and OMT. It largely draws from the renderings of each one of these methods, with a number of symbols thrown out (because they were confusing,

superfluous, or little-used) and with a few new symbols added.

From the outside, the Unified Modelling Language encompasses the following models:

Use-case diagrams

Class diagrams

State-machine diagrams

Message-trace diagrams

Object-message diagrams

Process diagrams

Module diagrams

Network diagrams

Use-case diagrams look pretty much as they do in Objectory.

Class diagrams look much like OMT class diagrams (classes are rendered as rectangles) but with most relationships drawn from Booch.

State-machine diagrams, as developed by David Harel, are the same as in Booch and OMT.

Message-trace diagrams were in all three methods (and have thus been unified). Object-message diagrams are derived from Booch (with the change that objects are no longer clouds, but structured clouds; we also adopt the numbering convention from Fusion). Process diagrams are a new invention, and module diagrams and network diagrams derive from Booch.

Thus, if you are a Booch user, you'll need to get over the use of clouds (rectangles are more space-efficient), but pretty much things look the same. If you are an OMT user, you'll see a number of niggling changes (such as the multiplicity balls replaced with

more explicit expressions) but no fundamental changes (though there are a number of additions). If you are an Objectory user, you'll see new notation (because Objectory didn't deal with much graphical modelling post analysis). For all three kinds of users, there will be new things to learn (such as stereotypes and properties) but nothing that a few minutes of reading won't do.

3.2.4 The process associated with the Unified Modelling Language (Rational Software)

It's architecture-driven, and it's incremental and iterative.

Booch, Objectory, and OMT all have well-defined processes, and these are indeed sufficient for the Unified Modelling Language. Beyond that, we don't yet specify a process. Our focus first is on a common, standard modelling language. Processes by their very nature must be tailored to the organization, culture, and problem domain at hand. What works for one context (shrink-wrapped software development, for example) would be a disaster for another (hard-real-time, human-rated systems, for example).

This does not mean that we are blind to the importance of process. We do think we understand what such a process looks like, but it is unnecessary for us to bind the Unified Modelling Language to a particular process, because experience has shown that the modelling languages of Booch, Objectory, and OMT are amenable to a spectrum of different processes. Thus, we have kept in mind which artefacts are important for visualizing different aspects of a system under development and which artefacts are critical for controlling and measuring the progress of a development team. Doing so has been sufficient for us to know what to keep and what to throw out yet still permit diverse organizations to apply the Unified Modelling Language successfully.

3.3 Conclusion

In this chapter, Methodology, basic concept of Methodology and UML-based design methodology was provided. Object Oriented design and UML were deemed suitable for this research project in terms.

CHAPTER 4 CASE STUDY

This chapter contained all the information of Information System Architecture in an intelligent, semantic machine control system, which included the introduction, environment, implementation technology, design and evaluation of an intelligent, semantic machine control system. It provided the descriptions of this Architecture, and supported the use-case to model the architecture. Finally, it provided test and evaluation of architecture, which explained the implementation in detail.

4.1 Introduction of the Data Exercise Management System (or DEMS)

Due to the fact that compared to other industries, the fitness industry is lagging behind on incorporating Information Technology into its operation; it is necessary and current to offer IT in fitness management.

Some global companies like TechnoGym and Kaiser offer machines with pneumatic and electronic control but have no DEMS. This project designed architecture for DEMS.

The aim with the DEMS development is to establish software functionality that meets the needs of the various fitness and health industry stakeholders to enhance their respective services offered.

The DEMS comprising an Exercise Performance Monitor, Exercise Director and Exercise and Health Status Vector (EHS Vector) and The Exercise and Health Data Trading House (E&H Data Trading House) which will control all data transactions.

The data accumulated by the DEMS will allow a number of peripheral stakeholders

access to previously inaccessible exerciser data around which new business offerings can be developed. This relates particularly to medical aids that offer exercise incentives.

In this study, a novel dynamic resistance exercise machine with onboard data management was developed. This new technology that combined dynamic resistance exercising with the electronic capturing / managing of data drawn from the machine user optimizes exercise effectiveness for users. A paradigm was required to shift from single function exercise equipment to intelligent exercise equipment integrated within an IT based information network.

Information System's driven - pneumatically actuated exercise equipment is a forward-facing unit in this project. Onboard was the Data Exercise Management System known as DEMS. The DEMS communicated exercise and fitness data, via an Intelligence Trading House, to Doctors and Coachers. This project novelty lies in the exercises configured for individuals, which allows for seamless exercise routine, without the need for physically adjusting 'weights'. DEMS is an additional novelty offering useful empirical information on exercise status to interested third parties.

4.1.1 Outcomes

An information system has been developed properly

In this section, we focused on how to establish information system architecture to supply an intelligent, semantic machine control system. The pandect of information system architecture is shown in Fig 4.1.

The information system includes five parts: client, application server, database server, built-in system and machine.

1. *Client is a user interface application which includes exerciser application, coach* application, system administrator application and system engineer application.
2. Application server offers a series application to supply user control the machine and catch data. It also has a user database for the module of security.
3. Database server has two databases, one is exercise database to store the data of exercise and the other is plan database to store the plan which coach made for exerciser.
4. Built-in system offers basal function to provide data to database server and application server, catch and output signal through Daq card to machine, it also has a module to show his bodily condition to the exerciser who is exercising in this machine.
5. Machine incept signal from Daq card via actuator to control action of machine, output signal to Daq card through sensor to catch position of movement.

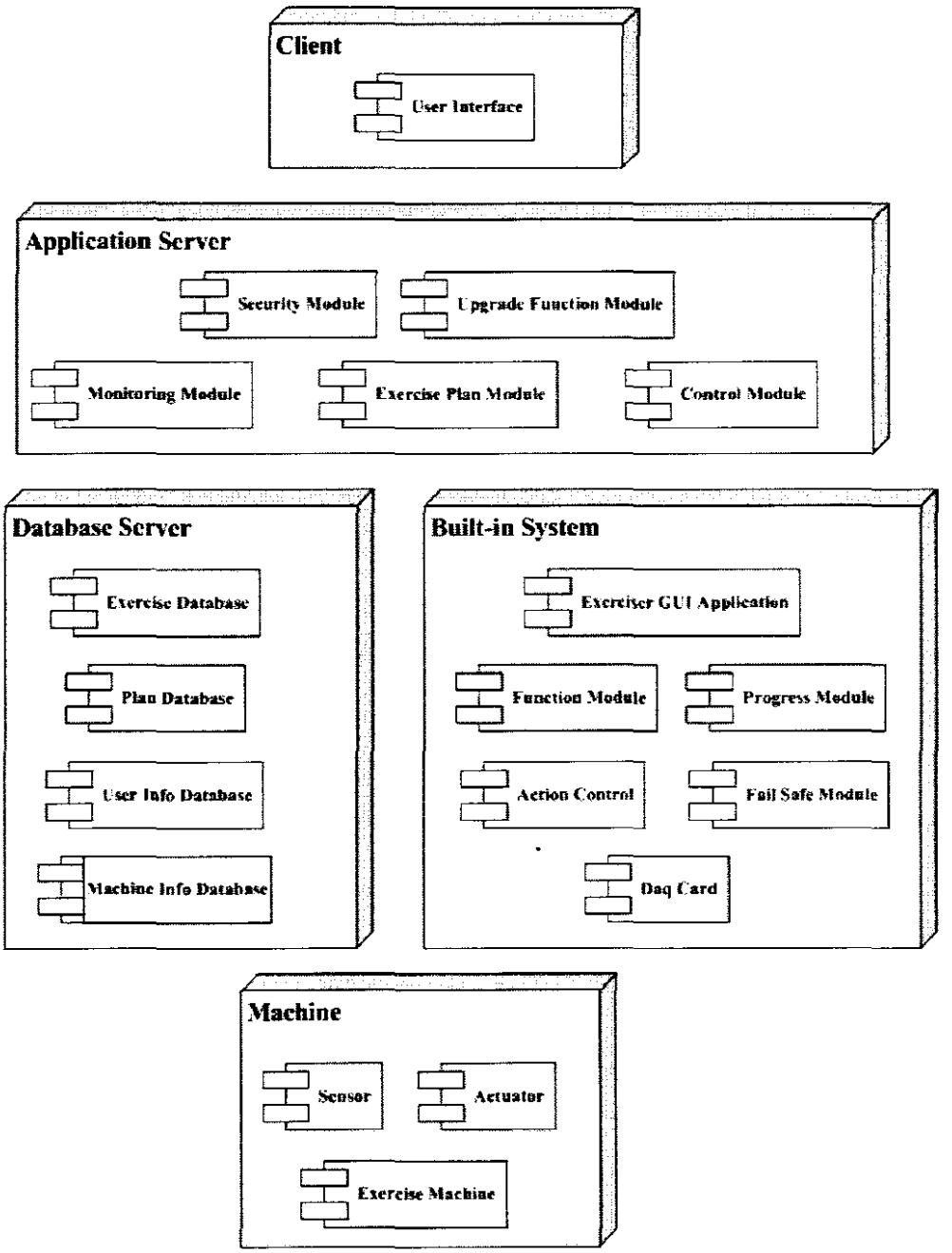


Fig 4.1 The summary of information system architecture

4.1.2 Anticipated Outputs

With the Exerciser and DEMS, an exerciser can be assessed after which a tailored program can be provided where the system monitors progress (Exercise Performance Monitor) and promotes program adherence (Exercise Director). The fitness consultant can then review the data (Exercise & Health Status Vector) ensuring that the user

meets the required fitness objectives. The remote monitoring and exercise prescription capacity will free fitness consultants from one-on-one monitoring, allowing them to expand their client-base, while ensuring that individualised scientific-based exercise and fitness goals are tracked. Currently, fitness consulting is the fastest growing sector in the health and fitness industry. No other product currently on the market offers fitness consultants this capacity.

The compact format of the FX control will allow for decentralisation of facilities and the establishment of new mini-gym facilities. These can be placed strategically in proximity to work places, at sports facilities, hotels, etc.

The data accumulated by the DEMS will allow a number of peripheral stakeholders to access to previously inaccessible exerciser data around which new business offerings can be developed. This relates particularly to medical aids that offer exercise incentives. However, these cannot be adequately managed currently, due to a lack of empirical data.

4.2 Environment

The physical exercise unit can be visualised as a stronger person facing you (who you are exercising your upper-body against) by taking hold of their hands or lying beneath you (holding on to your feet) against whom you exercise your legs. This design is indicated in Fig 4.2.

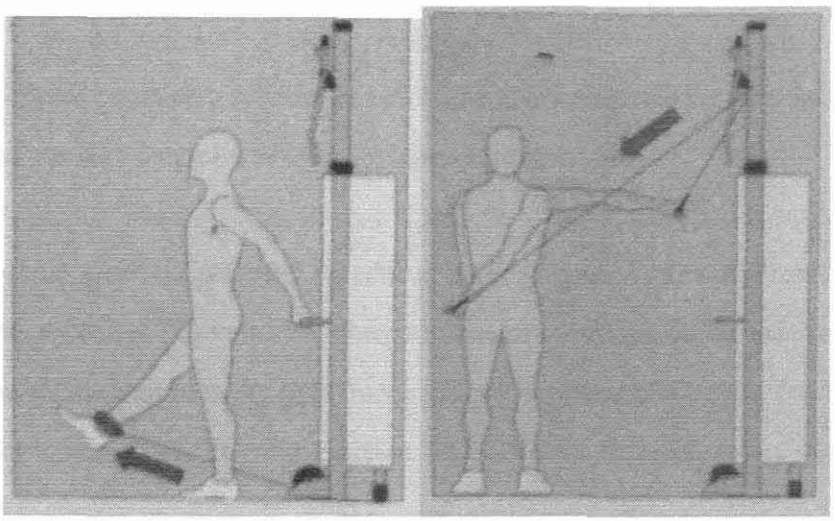


Fig 4.2 The physical exercise unit

4.2.1 The Electro-pneumatic Control Component

The FX arm and leg components are controlled by software through electro-pneumatic direct control. Similar technology is applied in industry to accomplish high-speed and accurate positioning in repetitive movement.

The principle known as FX-control relies on Differential Pressure Control as represented in Fig 4.3. FX-control can be described as follows:

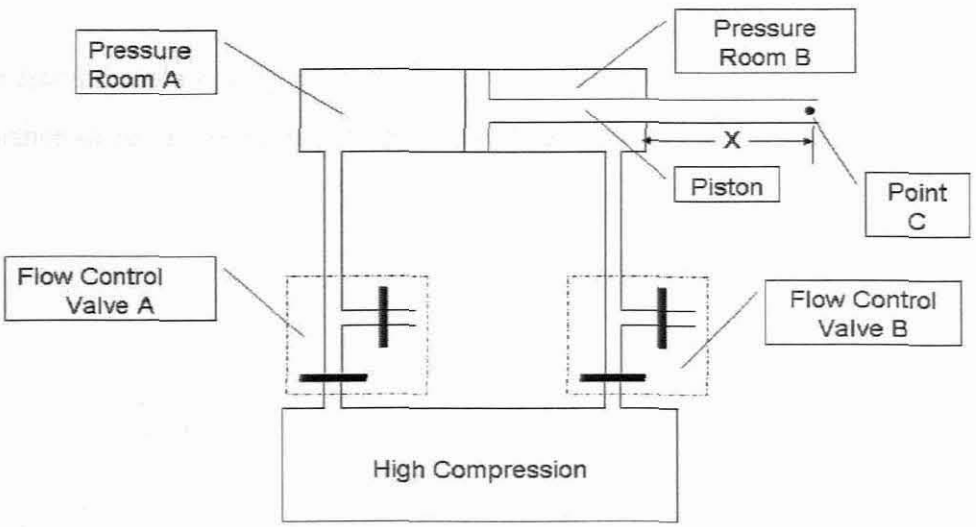


Fig 4.3 – Differential Pressure Control Principle

A bi-directional pneumatic cylinder has each chamber connected via an electronically controlled pressure control valve to a high pressure reservoir.

Pressure room A can be increased or reduced by Flow control valve A, Pressure room B also can be changed by valve B. When pressure of room A is higher than room B, the piston will be moved to the right by the margin of pressure. Thus the force can be exported via the piston. The valve A and valve B incept a different voltage signal from Daq card to control the different force.

The situation of the piston must be measured by a distance sensor. The sensor must be fixed in the piston, thus it is able to catch the situation of the piston relative to the pressure room. The sensor will export a different voltage signal to the Daq card when it is different situation.

In FX this is specifically controlled using a computer-based profile that determines the differential pressure in the in the cylinder relative to the spatial position of the mechanical linkage. An Exercise Trajectory Control System uses mathematical algorithms to deliver the required pressure to achieve the desired force trajectories.

We establish a physical structure to control pneumatic to export force and catch the situation of piston, the relation of the physical part is shown in Fig 4.4.

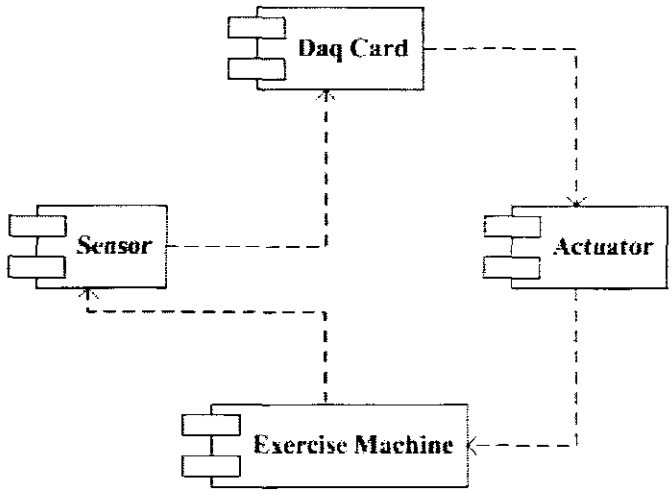


Fig 4.4 The relation of physical part

4.2.2 The Formulation of FX-control

For exercise special muscle with dynamic resistance exercising, a formulation of FX-control must be established. It supplies a function to exercise special muscle through offer difference force at difference situation.

At first a series of mode should be developed to supply exercise special muscle for different exercisers, this work must be done by coacher, doctor etc. It is able to be upgraded when other new mode is developed. In this case, the possible mode is *A()*, the general formula of FX-control is:

$$F() = A(x) \tag{1}$$

where *F()* is the exported force,

x is the real-time position of the arm or leg which needs to be determined.

Then for individual, according to different intensities and ranges of exercisers, some parameters should be considered. Therefore, the formula of FX-control now is:

$$F() = C + B \times A(x/y) \tag{3}$$

where *F()* is the exported force,

C is the initial force which can be given by coacher or exerciser himself,

B is a multiple quotiety,

A(x/y) is the exercise mode given by coacher or doctor,

x is the real-time position of arm or leg which needs to be determined,

y is the maximum position of arm or leg which is a known parameter.

The derivation of equation (3) is as following:

As mentioned above, equation (1) is:

$$F() = A(x) \tag{1}$$

To consider different exercise ranges, a real-time position of arm or leg (x) is changed to be a comparative position (x/y).

Substitute (x/y) to equation (1) to get equation (2):

$$F() = A(x/y) \tag{2}$$

To consider different exercise intensities, the mode *A()* will multiple a quotiety (B) and plus a initial force (C) to get exported force *F()*. Thus equation (3) becomes as:

$$F() = C + B \times A(x/y) \tag{3}$$

In this case, one formula also can be used repeatedly and continuously, it will be shown in the way as *D-F()*, where D is the number of repeating one action.

In one exercise, different formula of FX-control is able to be used. In a plan of exercise, the formula is listed below:

$$D_1 - F_1() = C_1 + B_1 \times A_1(x/y)$$

$$D_2 - F_2() = C_2 + B_2 \times A_2(x/y)$$

...

$$D_n - F_n() = C_n + B_n \times A_n(x/y) \tag{4}$$

The formula of mode must be installed in Function module, the function module is an upgradeable module, and the mode of exercise can be increased.

4.3 Implementation Technology

In this project, machine control system and information system architecture were developed. For different system, different platform were used, LabView was used to machine control system and .NET was used to information system. In this section, basic information about LabView and .NET were provided.

4.3.1 LabView (Using Graphical Programming Throughout the Development Cycle with NI LabVIEW 2005)

LabView is a highly productive graphical development environment with the performance and flexibility of a programming language, as well as high-level functionality and configuration utilities designed specifically for measurement and automation applications.

In general-purpose programming language, the code is as much of a concern as the application. In contrast, with LabView icons are used to represent functions, and they are wired together to determine the flow of data through our program, similar to creating flowcharts. It has all the breadth and depth of a general-purpose programming language, but it is easy to use, increasing the productivity by decreasing the time required to develop the applications.

It is easy to divide measurement and automation application into three main parts: acquisition, analysis, and presentation of data. LabView provides a seamless way to acquire the data, perform necessary analysis on that data, and present the information in a chosen format.

Each program in LabView is called a virtual instrument, or VI serves as the primary building block of a LabView application, and it can be used to modularize the code for efficient design, clear and concise documentation, and simplified maintenance. Each LabView VI is made up of two main components: the Front Panel and Block Diagram.

NI LabView Environment (National Instruments a)

The Front Panel is where the user interface is created for the VI. A Front Panel is built by dragging and dropping controls and indicators from the Control Palette. Each palette icon represents a sub palette, which contains the controls and indicators placed on the Front Panel. The controls and indicators are configurable and enable the user to create professional graphical interface. All of these objects were designed specifically for measurement applications. A control is Front Panel object that the user manipulates to interact with the VI. Simple examples of controls include buttons, slides, dials, and text boxes. An indicator is a Front Panel object that displays data to the user. Examples of indicators are graphs, thermometers, and gauges. When a control or indicator is placed on the Front Panel, a corresponding terminal is placed on the Block Diagram.

The Block Diagram is the brain of the VI, it is the home for the code. In this window, the functions palette is used to create the measurement application code. Each palette icon represents a sub palette, which contains VIs and functions placed on the Black Diagram and wired together to create the code. A complete Block Diagram has a similar appearance to a flowchart.

The sub palettes provide everything needed in terms of the constructs and functions found in any programming language as well as functions that are unique to LabView. These palettes, specifically designed for measurement and automation applications,

truly differentiate LabView from more traditional text-based programming language, the efficiency necessary being given to rapidly and easily develop the applications.

Express VIs are available in LabView 7 Express to streamline the application development. There are over 40 Express VIs include in LabView that enable to create complete measurement programs in seconds. These VIs were created for the most frequently built application with productivity and efficiency needs in mind. The power of Express VIs is found in the property pages for each that can be individually customized simply by double-clicking them. This will significantly reduce the number of objects on the Block Diagram and the time needed to add additional functionality. Additionally, hundreds of standard VIs are available from the All Functions Palette, making LabView a complete programming language.

LabView is basically a dataflow programming language. This means that data flows from a data source to one or more sinks and propagates through the program. Unlike text-based development software, LabView, because of its dataflow capability, is not sequential and can execute multiple operations in parallel easily using an intuitive diagram representation.

LabView is a multithreaded programming environment, meaning that multiple operations can occur simultaneously without interfering with each other.

LAbView takes full advantage of current and emerging commercial technologies such as operating systems, communications buses, and major technology changes such as .NET. Additionally, LabView is a multiplatform programming language, which means the LabView source code which has been written on one platform can be taken and reused on any of the other supported platforms. The developed application runs without any modifications. The only exception is if the code contains operating system-specific calls that only execute on a given platform.

Virtual Instrumentation (National Instruments b)

A virtual instrument consists of an industry-standard computer or workstation equipped with powerful application software, cost-effective modular hardware such as plug-in boards with appropriate driver software, and the unit under test(UUT) and sensors, all of which work together to perform the functions of traditional instruments. Virtual instrumentation leverages the power of commercially available PC technology such as processors, memory, and I/O to create instrumentation tools. The virtual instrumentation paradigm transformed test, measurement, and automation applications from loosely coupled and often incompatible stand-alone instruments and devices to tightly integrated, high-performance measurement and automation systems.

More than just building the equivalent of a traditional instrument around a PC, virtual instrumentation is about redefining what an instrument is and empowering users to build powerful instruments and flexible measurement systems never before possible. By using virtual instrumentation, a tightly integrated measurement and automation application can be achieved, which incorporates many kinds of I/O, such as data acquisition, motion control, image acquisition, and distributed I/O. With traditional instrumentation, the integration of a system like this would be costly and time-consuming.

Virtual instruments represent a fundamental shift from traditional hardware-centred instrumentation system to software-centred systems that exploit the computing power, productivity, display, and connectivity capabilities of popular desktop computers and workstations. Although the PC and integrated circuit technology have experienced significant advances in the last two decades, it is software that provides the leverage to build on this powerful hardware foundation to create virtual instruments, providing better ways to innovate and significantly reduce cost. With virtual instruments, engineers and scientists build measurement and automation systems that suit their needs exactly (user-defined) instead of being limited by traditional fixed-function

instruments(vendor-defined).

Software is the cornerstone of a virtual instrumentation system. Its flexibility, combined with powerful modular hardware solutions, creates the ultimate in user-defined, scalable instrumentation systems.

4.3.2 Microsoft .NET Platform

.NET Platform

The Microsoft® .NET platform provides all of the tools and technologies needed to build distributed applications. It exposes a language-independent, consistent programming model across all tiers of an application while providing seamless interoperability with, and easy migration from, existing technologies (Microsoft .NET Framework Reviewers Guide).

The .NET platform is made up of several core technologies as shown on the Fig 4.5. These technologies are described in the following topics (MSDN training).

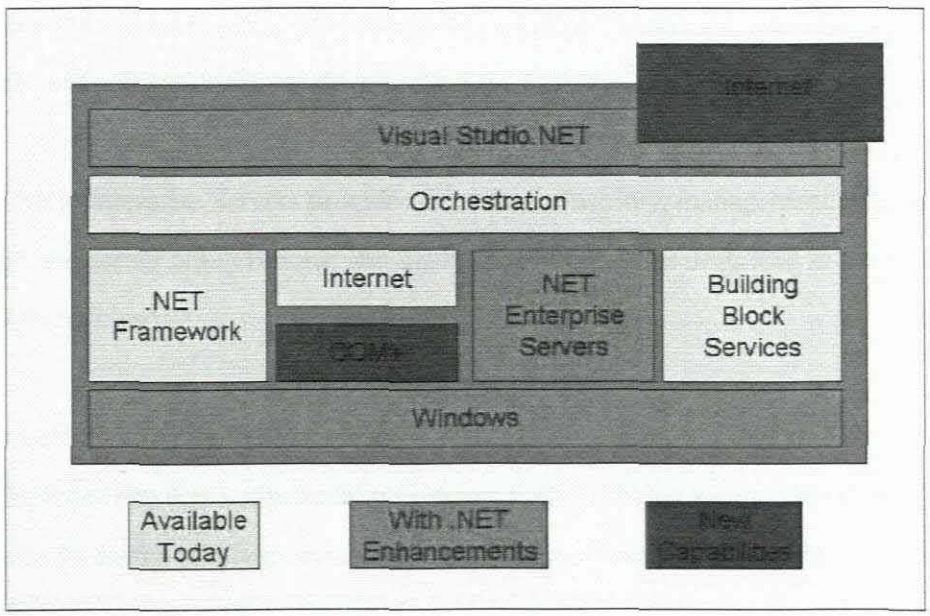


Fig 4.5 The .NET Framework (MSDN training)

The .NET Framework is based on a new Common Language Runtime. The Common Language Runtime provides a common set of services for projects built in Microsoft Visual Studio.NET, regardless of the language. These services provide key building blocks for applications of any type, across all application tiers (Vecchiola et al. 2002).

Microsoft Visual Basic®, Microsoft Visual C++®, and other Microsoft programming languages have been enhanced to take advantage of these services. Third-party languages that are written for the .NET platform also have access to the same services (MSDN training). The .NET Framework is explained in greater detail later in this module.

The .NET building block services are distributed programmable services that are available both online and offline. A service can be invoked on a stand-alone computer not connected to the Internet, provided by a local server running inside a company, or accessed by means of the Internet. Microsoft .NET building block services can be used from any platform that supports SOAP. Microsoft Windows-based clients are optimized to distribute Web Services to every kind of device. Services include identity, notification and messaging, personalization, schematized storage, calendar, directory, search, and software delivery (Microsoft .NET Framework Reviewers Guide).

The .NET Enterprise Servers provide scalability, reliability, management, integration within and across organizations, and many other features, as described in the below (MSDN training).

Microsoft SQL Server™2000

Includes rich XML functionality, support for Worldwide Web Consortium (W3C) standards, the ability to manipulate XML data by using Transact SQL (T-SQL), flexible and powerful Web-based analysis, and secure access to the data over the Web by using HTTP.

Microsoft BizTalk™ Server 2000

Provides enterprise application integration (EAI), business-to-business integration, and the advanced BizTalk Orchestration technology to build dynamic business processes that span applications, platforms , and organizations over the Internet.

Microsoft Host Integration Server 2000

Provides the best way to embrace Internet, intranet, and client/server technologies while preserving investments in existing earlier systems.

Microsoft Exchange 2000 Enterprise Server

Builds on the powerful Exchange messaging and collaboration technology by introducing several important new features, and further increasing the reliability, scalability, and performance of its core architecture. Other features enhance the integration of Exchange 2000 with Microsoft Windows 2000, Microsoft Office 2000, and the Internet.

Microsoft Application Centre 2000

Provides a deployment and management tool for high-availability Web applications.

Microsoft Internet Security and Acceleration Server 2000

Provides secure, fast, and manageable Internet connectivity. Internet Security and Acceleration Server integrate an extensible, multilayer enterprise firewall and a scalable high-performance Web cache. It builds on Windows 2000 security and directory for policy-based security, acceleration, and management of internetworking.

Microsoft Commerce Server 2000

Provides an application framework, sophisticated feedback mechanisms, and analytical capabilities.

Visual Studio.NET provides a high-level development environment for building applications on the .NET Framework. It provides key enabling technologies to simplify the creation, deployment, and ongoing evolution of secure, scalable, highly available Web applications and Web Services (Microsoft .NET Framework Reviewers

Guide).

The next generation of Microsoft Windows® will provide the foundation for developers who want to create new .NET applications and services. (Microsoft)

.NET Framework

Before COM, applications were completely separate entities with little or no integration. By using COM, components can be integrated within and across applications by exposing common interfaces. However, as a developer, we must still write the code to wrap, manage, and clean up after components and objects (.NET Framework Overview).

In the .NET Framework, components are built on a common foundation. We no longer need to write the code to allow objects to interact directly with each other. In addition, we no longer need to write component wrappers in the .NET environment, because components do not use wrappers. The .NET Framework can interpret the constructs that developers are accustomed to using in object-oriented languages. The .NET Framework fully supports class, inheritance, methods, properties, events, polymorphism, constructors, and other object-oriented constructs (.NET Framework Overview).

The Common Language Specification (CLS) defines the common standards to which languages and developers must adhere if they want their components and applications to be widely useable by other .NET languages (MSDN training).

In the .NET Framework, Visual Studio.NET provides the tools we can use for rapid application development (MSDN training).

The .NET Framework fully supports the existing Internet technologies including

Hypertext Markup Language (HTML), XML, SOAP, Extensible Stylesheet Language for Transformations (XSLT), Xpath, and other Web standards. The .NET Framework favors loosely connected, stateless Web services (MSDN training).

A .NET class's functionality is available from any .NET language or programming model (MSDN training).

In the .NET Framework, code is organized into hierarchical namespaces and classes. The Framework provides a common type system, referred to as the unified type system, that is used by any .NET language. In the unified type system, all languages elements are objects. There are no variant types, there is only one string type, and all string data is Unicode. The unified type system is described in more detail in later modules (MSDN training).

The hierarchy of the .NET Framework is not hidden from the developer. It can be accessed and extended .NET classes (unless they are sealed) through inheritance. Cross-language inheritance can also be implemented (MSDN training).

The .NET Framework Components

The .NET Framework is a set of technologies that form an integral part of the Microsoft .NET platform. It provides the basic building blocks for developing Web applications and Web services see Fig 4.6 (MSDN training).

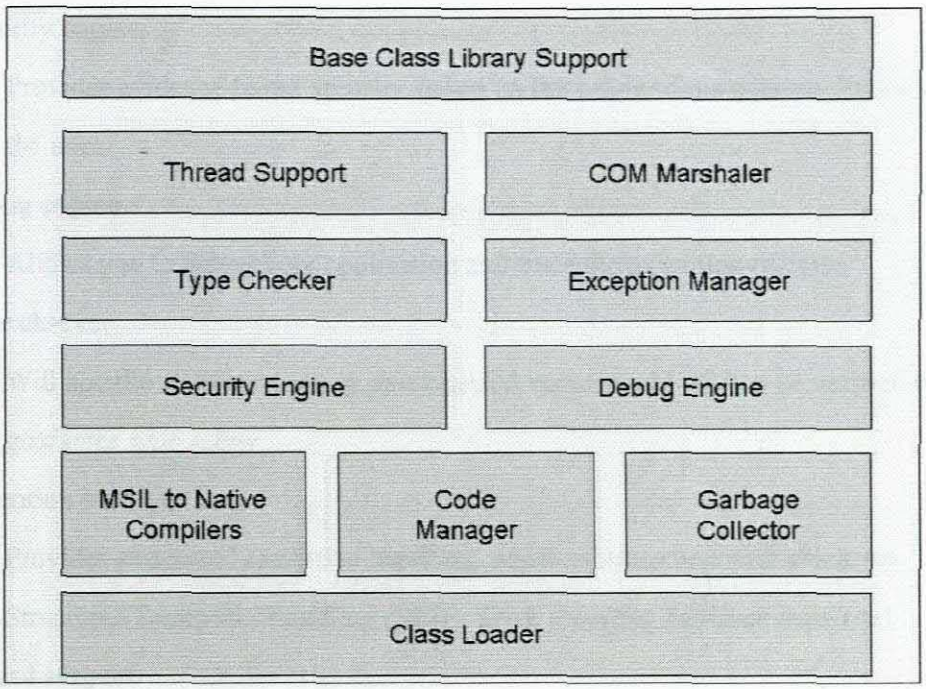


Fig 4.6 Common Language Runtime (MSDN training)

The Common Language Runtime simplifies application development, provides a robust and secure execution environment, supports multiple languages, and simplifies application deployment and management. The environment is also referred to as a managed environment, one in which common services, such as garbage collection and security, are automatically provided. The Common Language Runtime features are described below (.NET Framework Overview).

Class loader

Manages metadata, as well as the loading and layout of classes.

Microsoft intermediate language (MSIL) to native compiler

Converts MSIL to native code (Just-in-Time).

Code manager

Manages code execution.

Garbage collector (GC)

Provides automatic lifetime management of all of the objects. This is a multiprocessor, scalable garbage collector.

Security engine

Provides evidence-based security, based on the origin of the code in addition to the user.

Debug engine

Allows you to debug your application and trace the execution of code.

Type checker

Will not allow unsafe casts or uninitialized variables. MSIL can be verified to guarantee type safety.

Exception manager

Provides structured exception handling, which is integrated with Windows Structured Exception Handling (SEH). Error reporting has been improved.

Thread support

Provides classes and interfaces that enable multithreaded programming.

COM marshaller

Provides marshalling to and from COM.

Base Class Library (BCL) support

Integrates code with the runtime that supports the BCL.

The Base Class Library (BCL) exposes features of the runtime and provides other high-level services that every programmer needs through namespaces. For example, the System.IO namespace contains input/output (I/O) services (.NET Framework Overview).

In the System.IO namespace, all of the base data types, such as int and float, are defined for the platform. Inside the System.IO namespace, there are other namespaces that provide various runtime features. The Collections namespace provides sorted lists, hash tables, and other ways to group data. The IO namespace provides file I/O, streams, and so on. The Net namespace provides Transmission Control Protoc ol/Internet Protocol (TCP/IP) and sockets support. For more information about namespaces, search for "namespaces" in the .NET Framework SDK Help documents

(.NET Framework Overview).

System.WinForms Classes

The System.WinForms classes can be used to build the client user interface (UI). This class lets the standards Windows UI be implemented in the .NET applications.

System.Drawing Classes

The System.Drawing classes can be used to access the new GDI+ features. This class provides support for the next generation of Graphics Device Interface (GDI) two-dimensional graphics. It also provides native support for Graphics Interchange Format (GIF), Tagged Image File Format (TIFF), and other formats.

The .NET Framework provides support for several programming languages. C# is the programming language specifically designed for the .NET platform, but C++ and Visual Basic have also been upgraded to fully support the .NET Framework.

C#

C# is designed for the .NET platform and is the first modern component-oriented language in the C and C++ family. It can be embedded in ASP.NET pages. Some of the key features of this language include classes, interfaces, delegates, boxing and unboxing, namespaces, properties, indexers, events, operator overloading, versioning, attributes, unsafe code, and XML documentation generation. No header or Interface Definition Language (IDL) files are needed.

Managed Extensions to C++

The managed C++ is a minimal extension to the C++ language. This extension provides access to the .NET Framework that includes garbage collection, single-implementation inheritance, and multiple-interface inheritance. This upgrade also eliminates the need to write "plumbing" code for components. It offers low-level access where useful.

Visual Basic.NET

Visual Basic.NET provides substantial language innovations over previous versions of Visual Basic. Visual Basic.NET supports inherit ance, constructors, polymorphism, constructor overloading, structured exceptions, stricter type checking, free threading, and many other features. There is only one form of assignment? no Let or Set methods. There are new Rapid Application Development (RAD) features such as XML Designer, Server Explorer, and Web Forms designer available from Visual Studio.NET to Visual Basic. With this release, Visual Basic Scripting Edition provides full Visual Basic functionality.

Microsoft JScript.NET

JScript.NET is rewritten to be fully .NET aware. It includes support for classes, inheritance, types, and compilation. It provides improved performance and productivity features. JScript.NET is also integrated with Visual Studio.NET. You can take advantage of any .NET Framework class in JScript.NET.

Third-party languages

Several third-party languages are supporting the .NET platform. These languages include APL, COBOL, Pascal, Eiffel, Haskell, ML, Oberon, Perl, Python, Scheme, and SmallTalk.

4.4 Design

4.4.1 Understands the demand

This information system is the supplier system to the exercise machines, it works with exercise machine in the gym or a wide Domain, provides a series of information of exercisers to exerciser or coacher or third party.

1. System supplies exercise function to exerciser and exercise machine. Exerciser must register in this system.
2. System takes charge of managing exercise machine and exerciser, if the machine is broken, it will be cancelled.

3. Exerciser needs to monitor the state of exercise to catch his data of exercise.
4. Coacher needs to monitor the state of exercise to control the machine locally or remotely when exerciser is exercising, catch data of exerciser, and make a plan for the next time.
5. System administrator needs to admin machine and exerciser information.
6. System engineer needs to upgrade the function module.

4.4.2 The demand analyzes

In this system, the actors are Exerciser, Coacher, System administrator and System engineer.

The use case is shown below (see Fig 4.7 and Fig 4.8):

1. Exerciser is exercising on exercise machine.
2. Exerciser and coacher need to monitor the state of exercise locally when exercise is exercising.
3. Exerciser and coacher need to catch exerciser's data of exercise.
4. Exerciser and coacher need to make a plan for the next time.
5. Coacher needs to monitor the state of exercise remotely when exerciser is exercising.
6. Coacher needs to control the exercise machine real-time locally or remotely when exerciser is exercising.
7. System administrator needs to manage the information of user (exerciser, coacher, system administrator and system engineer) and the machine.
8. System administrator needs to catch the data of exercise for exerciser.
9. System engineer needs to upgrade the function module.

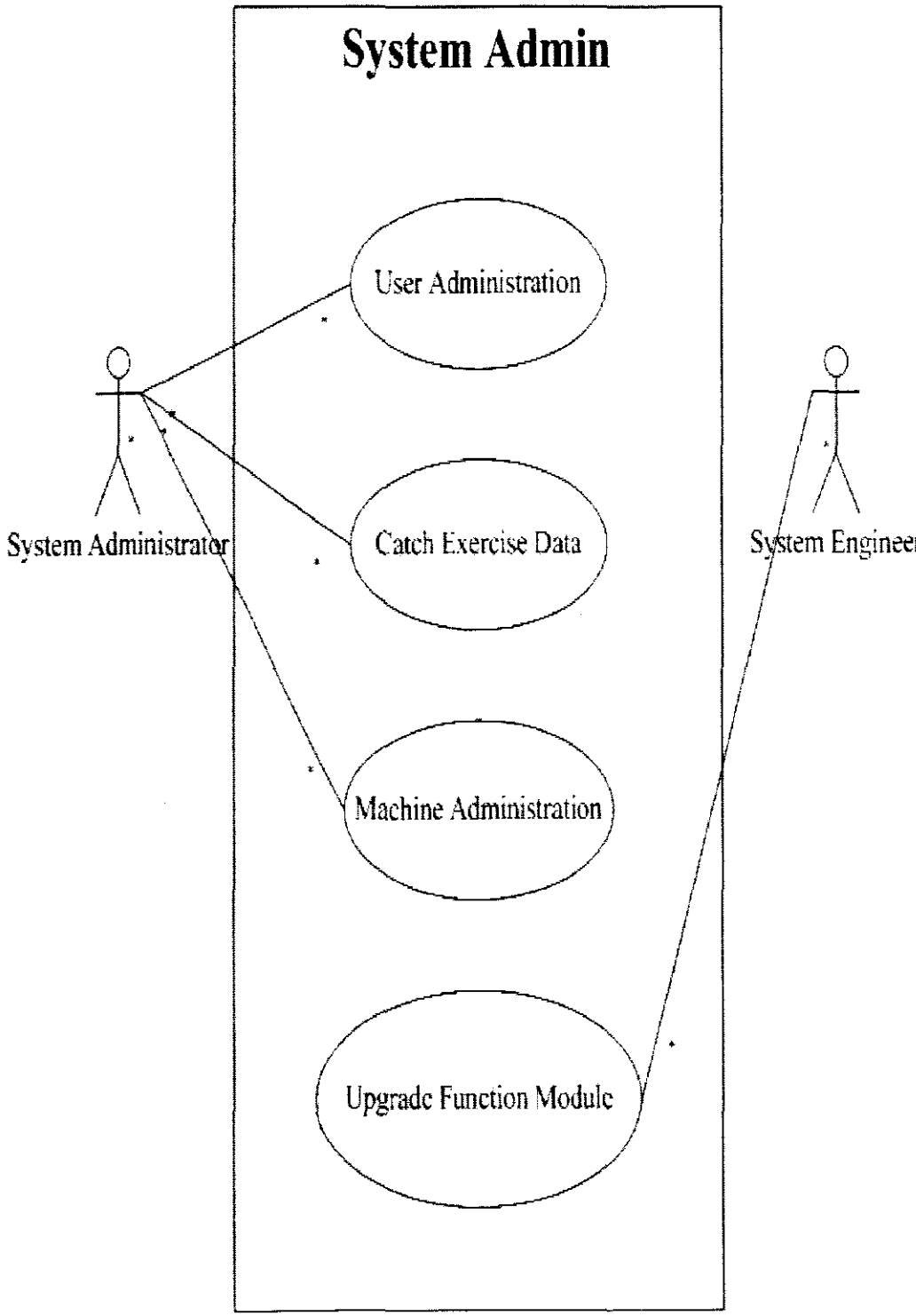


Fig 4.7 Use case view of system admin

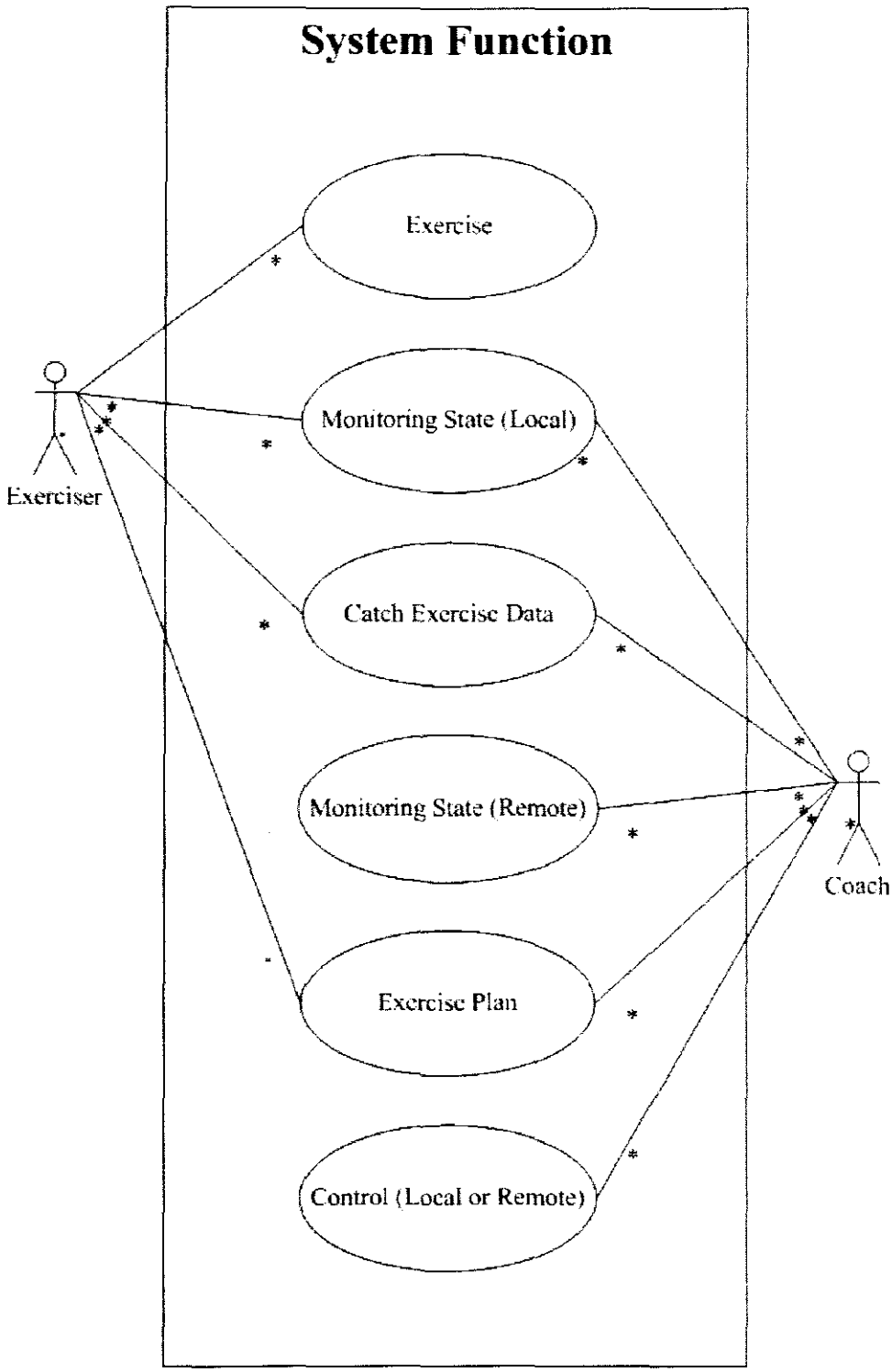


Fig 4.8 Use case view of system function

The user of this system can work on the client (computer) or the exercise machine: the work of system admin can be done on the client. Exerciser can do exercise and monitor the state of exercise on the exercise machine, catch data and make exercise

plan on the client. Coacher can do his work on the client but control and monitor locally.

The use case of exercise is described in Fig 4.9.

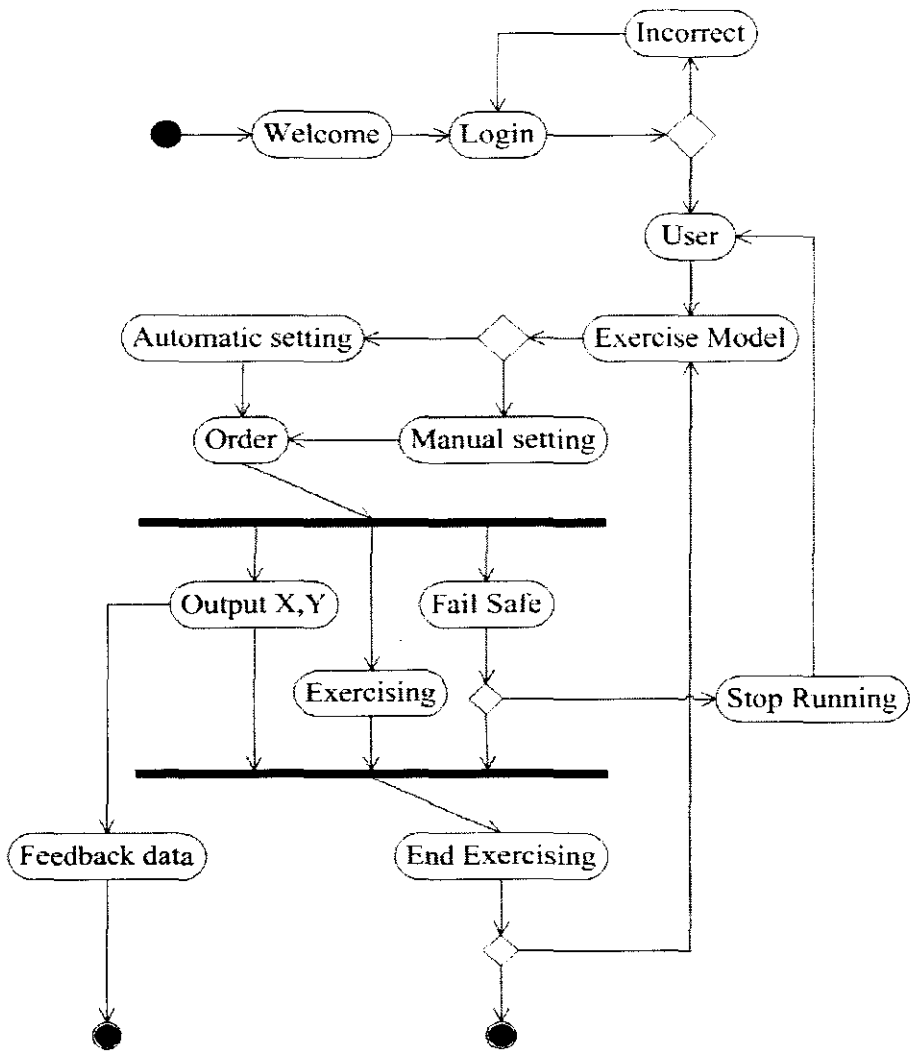


Fig 4.9 Exercise activity diagram

When exerciser exercises, the coacher is able to monitor the state of exercise from the screen which is built in the exercise machine. The coacher also is able to control the exercise machine, that means coacher can change parameter A, B, C, D in the formula *D–F()=C–B*A(x y)*.

When a user is on client, four modes can be chosen, exerciser, coacher, system administrator and system engineer. A user can log in the system by default actor if he has another actor, but he can change it.

The use case for a user on client is described roughly in Fig 4.10.

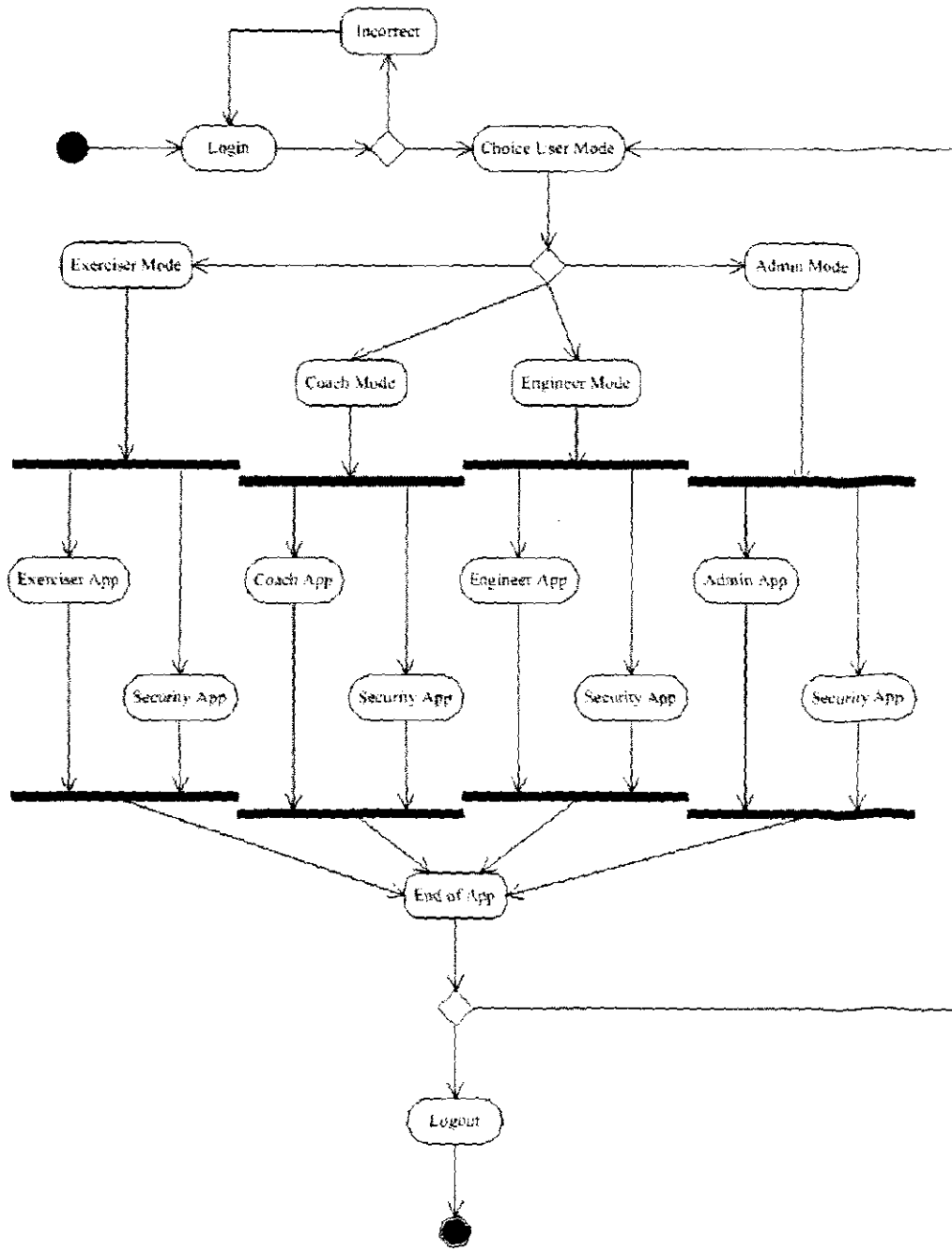


Fig 4.10 User activity diagram

In the diagram above, the exerciser app(application) can be run when a user chooses exerciser mode; whilst the coach app(application) can be run when choosing coach mode; In the same way, the admin app(system administrator application) and the engineer app(system engineer application) can be run when choosing the corresponding mode respectively.

When the user operates under the application mode, the security app(security application) is also running, which checks the user has the right to running application or not. If the answer is negative, the application mode will be stopped and turn back to the step of choice user mode.

In this system, exerciser can read his own data, modify the detail for himself and checks the exercise data, but modificative the exercise data is not allowed. Coacher can read exercise data of his own exerciser only. Exercise data can be checked, but modification is not allowed. The exercise plan can be made by exerciser himself or his coacher.

In this system, a system administrator has the right to add, delete a user and modify the detail of user; and the exercise machine read the statistical data of exercise but not exercise data of special exerciser. A system engineer has the right to upgrade function module to plus new formula into system.

The work flow of exerciser mode is described roughly in Fig 4.11.

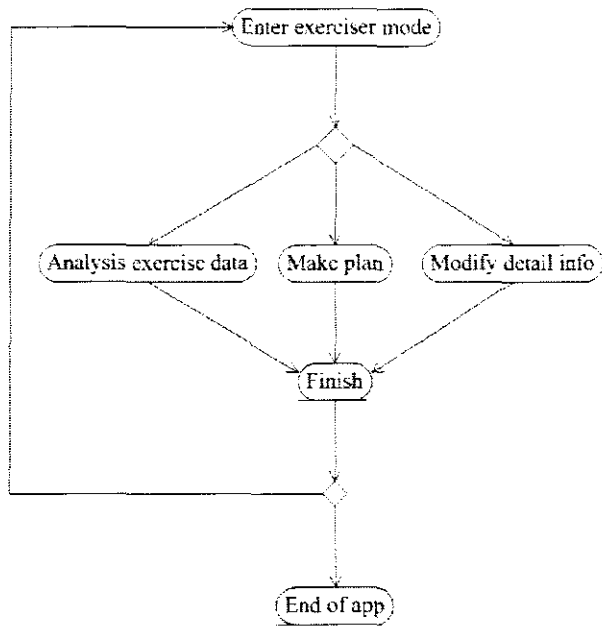


Fig 4.11 Work flow of exerciser mode

The work flow of coacher mode is described roughly in Fig 4.12.

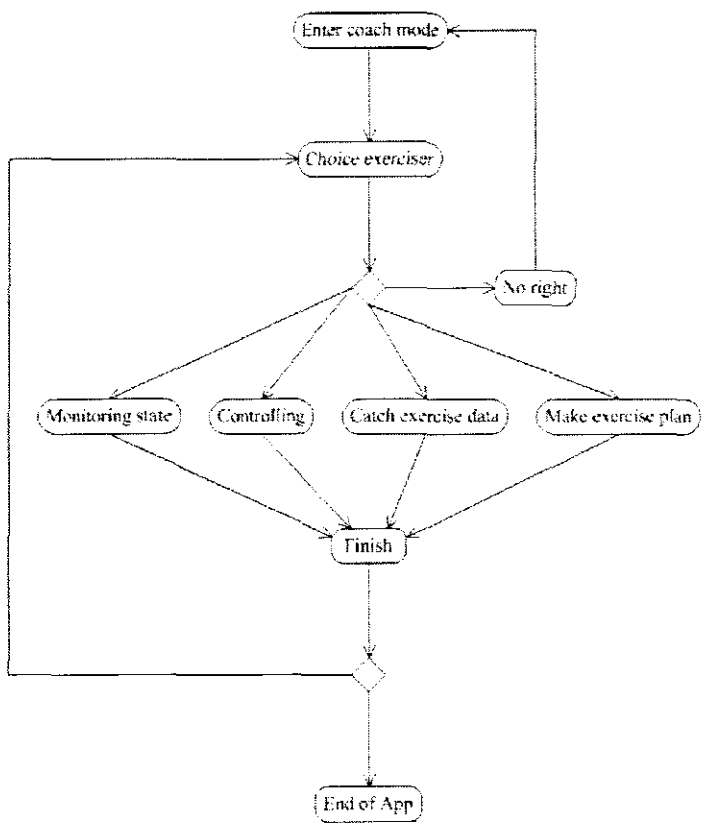


Fig 4.12 Work flow of coach mode

The work flow of admin mode is described roughly in Fig 4.13.

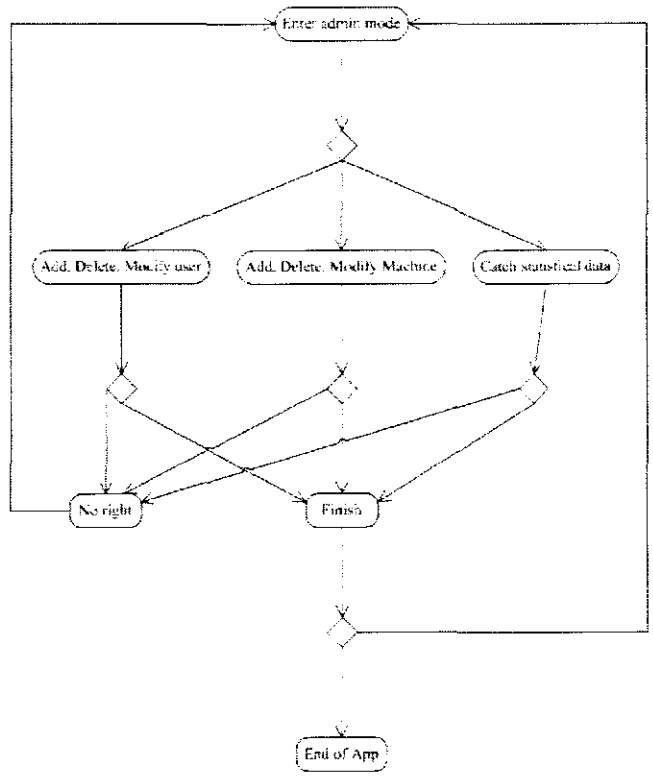


Fig 4.13 Work flow of admin mode

The work flow of engineer mode is described roughly in Fig 4.14.

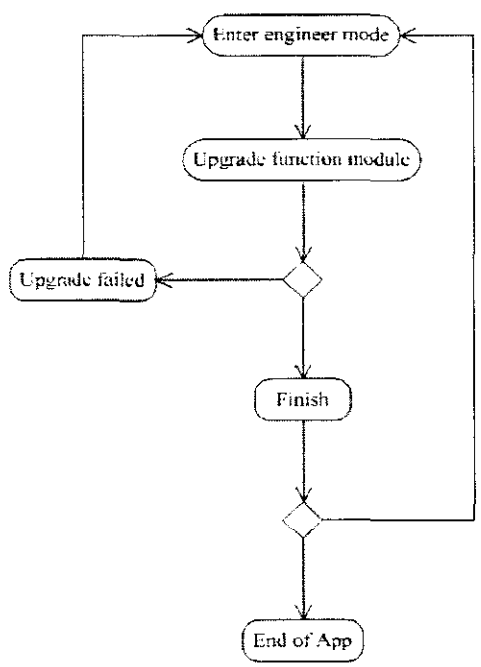


Fig 4.14 Work flow of engineer mode

4.4.3 Class Design

In this system, the basic function was to supply a novel dynamic resistance to the exerciser; hence the physical machine must be done before we developed an information system. The machine was controlled by an actuator, and returned to its state by sensor. In the information system physical machine, the actuator and the sensor were put together in one class which is "Machine". For checking the state of machine, it was necessary to know the position and force for transferring the information which included two functions, one was GetPosition(), and the other was InceptForce(). To control the machine to output correct force, function ForceControl() must be included. Therefore, there were two attributes and two operations in this class which is shown in Fig 4.15 by UML format.

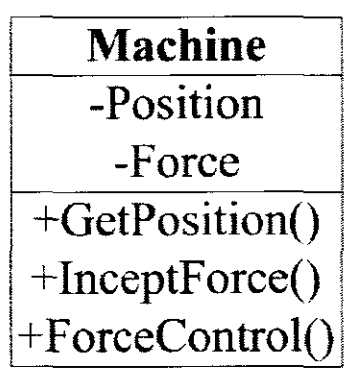


Fig 4.15 The class of Machine

In fact, class "Machine" was an abstract describe, included the sensor, the actuator and exercise machine, the data in this class was value of voltage. It took charge to measure the position of machine, returned a voltage value to Daq card, and incepted a voltage value from Daq card to control force of machine.

The class of Daq converted voltage value to data of force to control physical machine (class machine), and converted a data to voltage value of position to a data. So two operations must be included, ForceControl() and PositionIncept(). This class was

programmed by LabView, which is shown in Fig 4.16 by UML format.

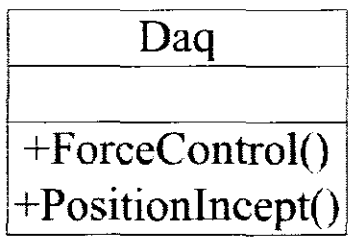


Fig 4.16 The class of Daq

The class of action took charge of receiving the data of position and calculate how much force must be transferred to class Daq by using formula *F()=C+B*A(x,y)*. Action stored the data of position and force in exercise database. Therefore, four operations, i.e. Compute(), Position(), Force() and Store() were included. And three parameters A, B, C were attributed. The class is shown in Fig 4.17 by UML format.

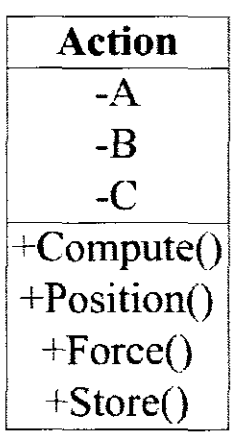


Fig 4.17 The class of Action

The class of FailSave took charge of keeping the exerciser safe when he was exercising. Class FailSave determined whether it was danger or not by calculating the speed of movement. When danger was coming, class FailSave sent a stop signal to class Daq immediately and Daq stopped output force at once. The class is shown in Fig 4.18 by UML format.

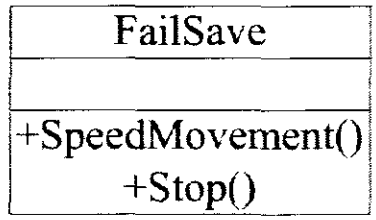


Fig 4.18 The class of FailSave

The class of function which was upgraded by system engineer had a series of formula to supply action. In this class, an operation Formula() which was called as *Function.Formula(A,B,C)* by class Action, was used to supply formula. To supply upgrading by system engineer was another operation of function. The class is shown in Fig 4.19 by UML format.

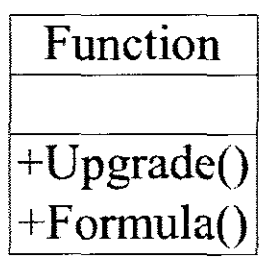


Fig 4.19 The class of Function

The class of progress received the order of action from exerciser GUI application, and then transferred it to a list of action, which called class "Action" to control the exercise machine and repeated it by different after one action was finished. The list of action which was stored in an array called "List" was also changed by class "Control", so this class had two operations: Action() and Control(). To send the list of action to Monitoring class, there was another operation ShowList(). The class is shown in Fig 4.20 by UML format.

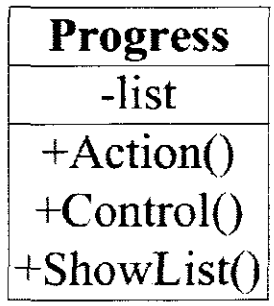


Fig 4.20 The class of Progress

The class of GUI was the interface of the exercise machine by which exerciser ordered the exercise. GUI showed the state of exercise to exerciser when he was exercising and called *Action.Position()* to catch the data of state. The list of action was private, so it was not changed directly and must call operation *Progress.Control()*. This class included operation "Order()", "Show()" and "ListChange()". The class is shown in Fig 4.21 by UML format.

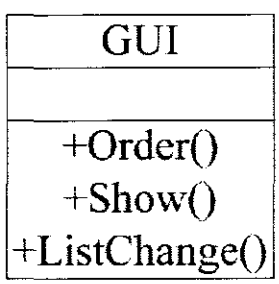


Fig 4.21 The class of GUI

It was not necessary to store the data in database and had a security control for a stand alone system; so the use of above classes was enough to order to the exercise mode on machine to exercise for exerciser. The relation of the classes in a stand alone system is shown below.

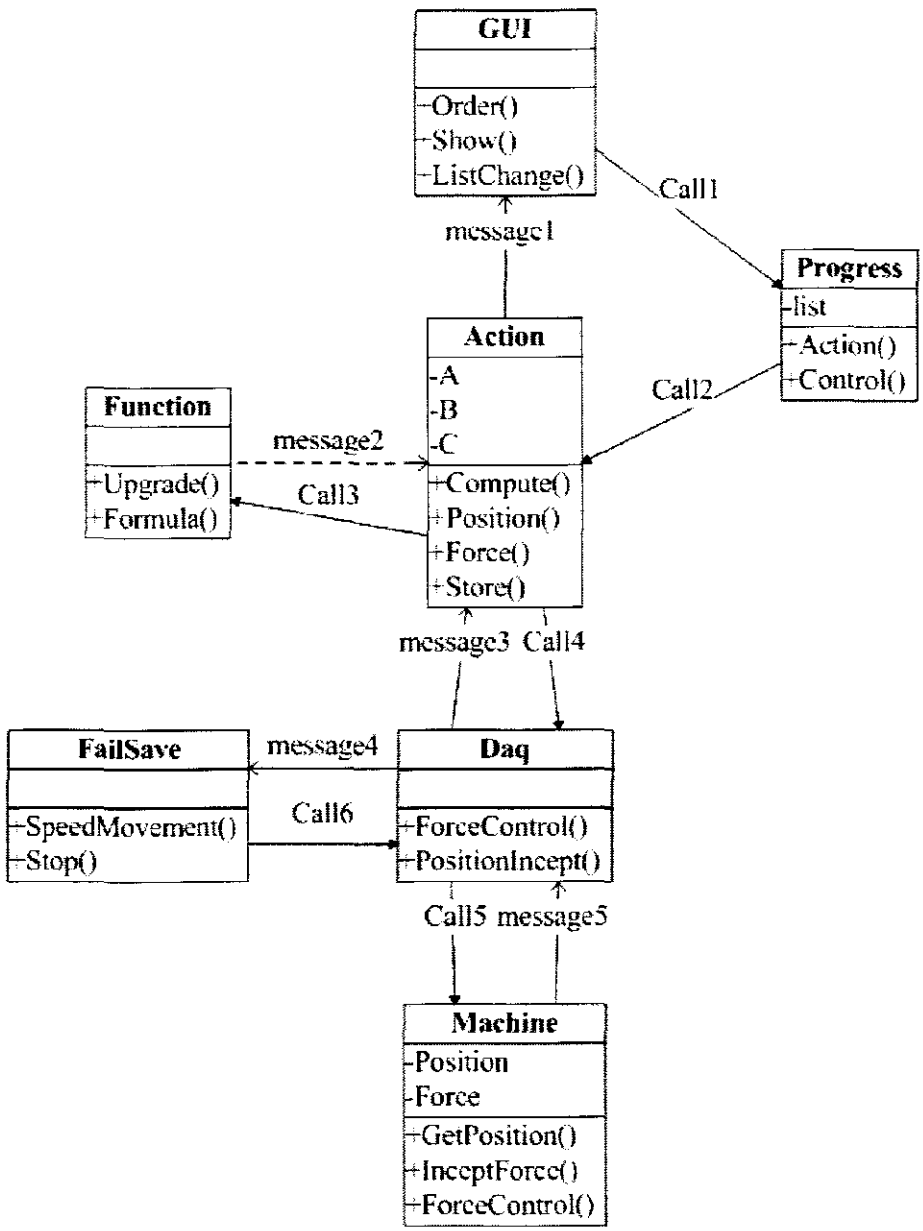


Fig4.22 The relation of the classes in a stand alone system

In the stand alone system, each function was one box which included exercise machine, sensor, actuator, screen and keyboard etc. This system was divided into two nodes, one was machine and the other was built-in system. In the part of machine, exercise machine, sensor and actuator were inside; they were showed in class of "Machine". In the part of built-in system, there were six classes: Daq, FailSave, Action, Function, Progress and GUI, which were called components. While in machine, there were three components: exercise machine, sensor and actuator.

Different component had different function, they called each other and transfer message. The physical view of stand alone system is shown in Fig 4.23. This system was also divided into three levers: physical lever, control lever and user lever. The lever structure is shown in Fig 4.24.

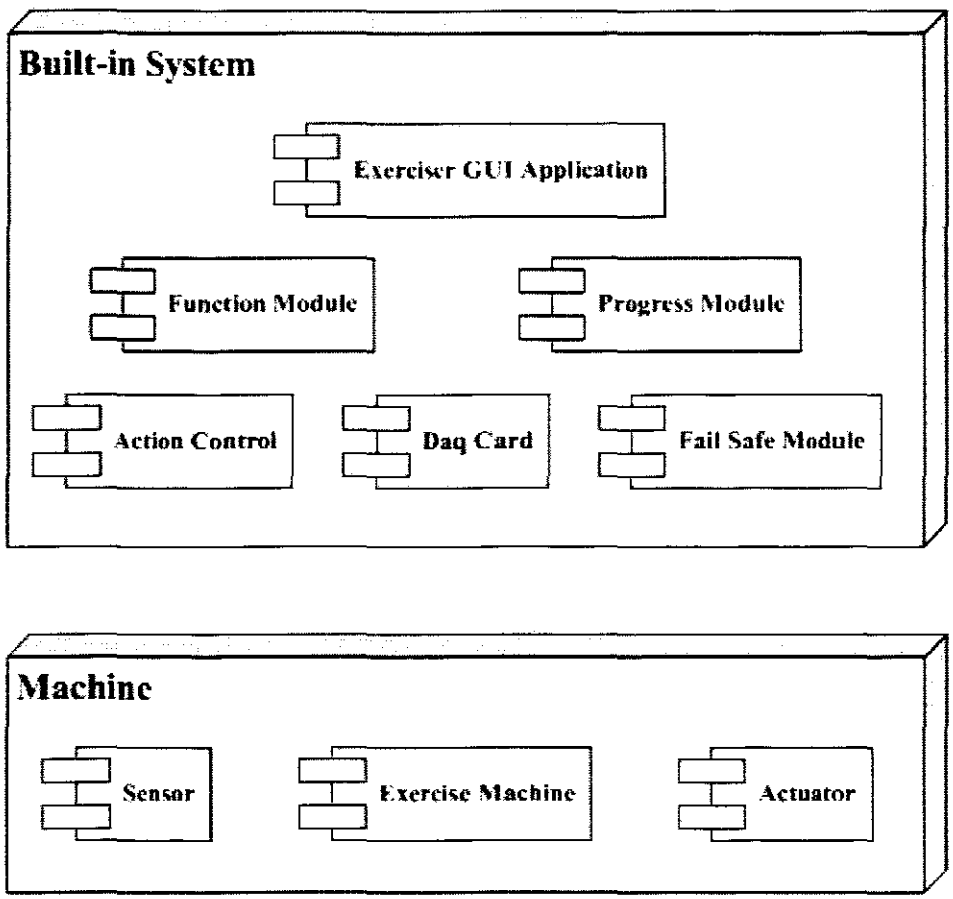


Fig4.23 The physical view of stand alone system

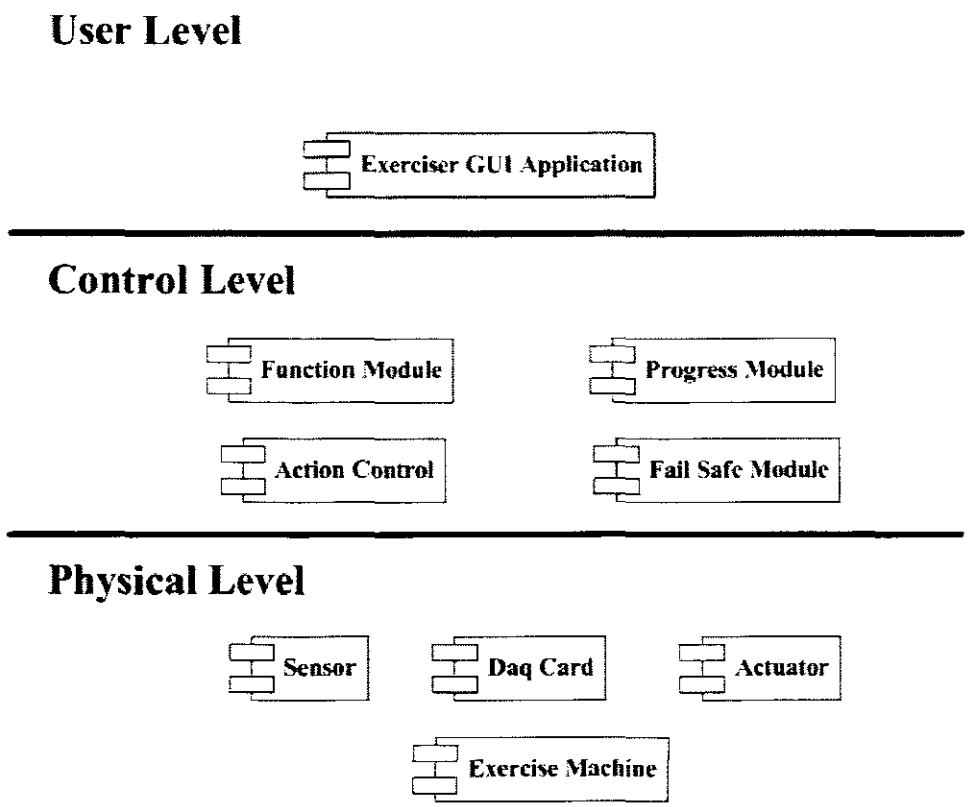


Fig 2.24 The level structure of stand alone system

In this system, the detail of machine and the detail of user were stored. In class DetailMachine and DetailUser, three operations were included, StoreDetail(), ModifyDetail(), and GetDetail(). In class DetailMachine, information about machine ID, location, model, edition, state(running, stand by, switch off) were stored and obtained. In class DetailUser, information about user ID, private information and right list were stored and obtained. The class is shown in Fig 4.25 by UML format.

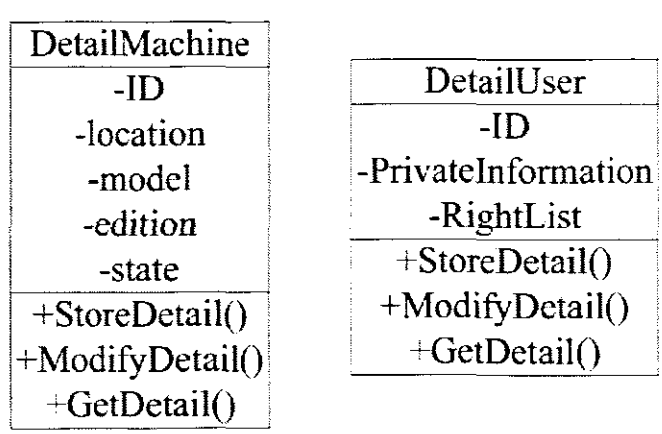


Fig 4.25 The class of DetailMachine and DetailUser

For security application, it always ran on the application server, by which any request from client was checked. The security application returned the data what was requested to the client if the request passed, otherwise, it returned a warning. It ensured right action was done by right user. No user can get any information what he was not allowed to touch. For example, an exerciser had the right to touch his own information about exercise, but he could not touch any other exerciser's information; a coach had the right to check the exercise information of his own exerciser, he could not touch any other exerciser's information. In security class, a message was returned to main application (exercise application, coach application, system admin application, system engineer application), if the request was allowed. So function IsAllowed() must be included. There was a private operation, Check(), which would be called by IsAliowed() to check the request was allowed, then returned yes or no to operation IsAllowed(). The class is shown in Fig 4.26 by UML format.

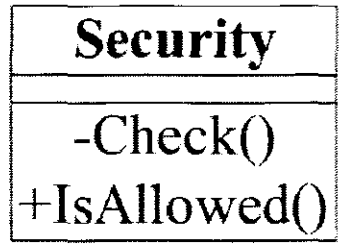


Fig 4.26 The class of security

An exercise plan class was needed to make the plan for next exercise. Exerciser did it himself, and coacher did it for his own exerciser. When the plan was done, it was stored to database. So in this class, three operations were included: PlanMake() and PlanStore(). Sometime the plan needed to be changed, so the other operation, PlanChange(), was also included. The class is shown in Fig 4.27 by UML format.

Progress.control() to change the list of action in the class of progress. Therefore an operation ChangList() was needed. The class is shown in Fig 4.30 by UML format.

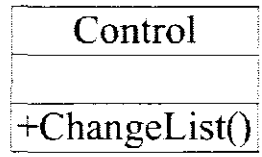


Fig 4.30 The class of Monitoring

An admin class was also needed to manage the user and information. This class inserted, deleted, upgraded the information of user and machine, and collected the information of exercisers. The class is shown in Fig 4.31 by UML format.

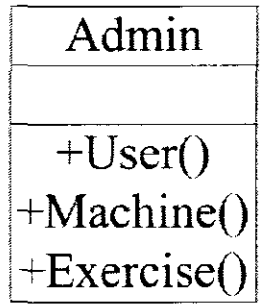


Fig 4.31 The class of Admin

Finally an interface class was needed to show everything to user. This class called a series of operations above mentioned. The class is shown in Fig 4.32 by UML format.

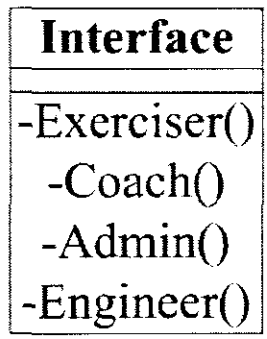


Fig 4.32 The class of Interface

In the Unified Modelling Language, a node corresponding to hardware or a piece of a larger hardware component, represented by encapsulating the node (having one node inside the other node). The node contained a stereotype that represented what the hardware was used for. The attributes of the node represented the hardware's configuration.

Components were corresponding to software. Each component was representative of one or more classes that implement one function within the system. A component could represent something as granular as a price adapter, to a sub-system to handle a trade entry.

In fact, each class of the above from class of machine to class of interface was regarded as component. In machine, there were three components: exercise machine, sensor and actuator. In the stand alone system, two nodes and three levels had already been created, while in whole system, five nodes (See Fig 4.1) and five levels were created.

The whole system was also divided into five levels according to different effects: user level, Application Level, Data Level, Control Level and Physical Level. The lever structure is shown in Fig 4.33.

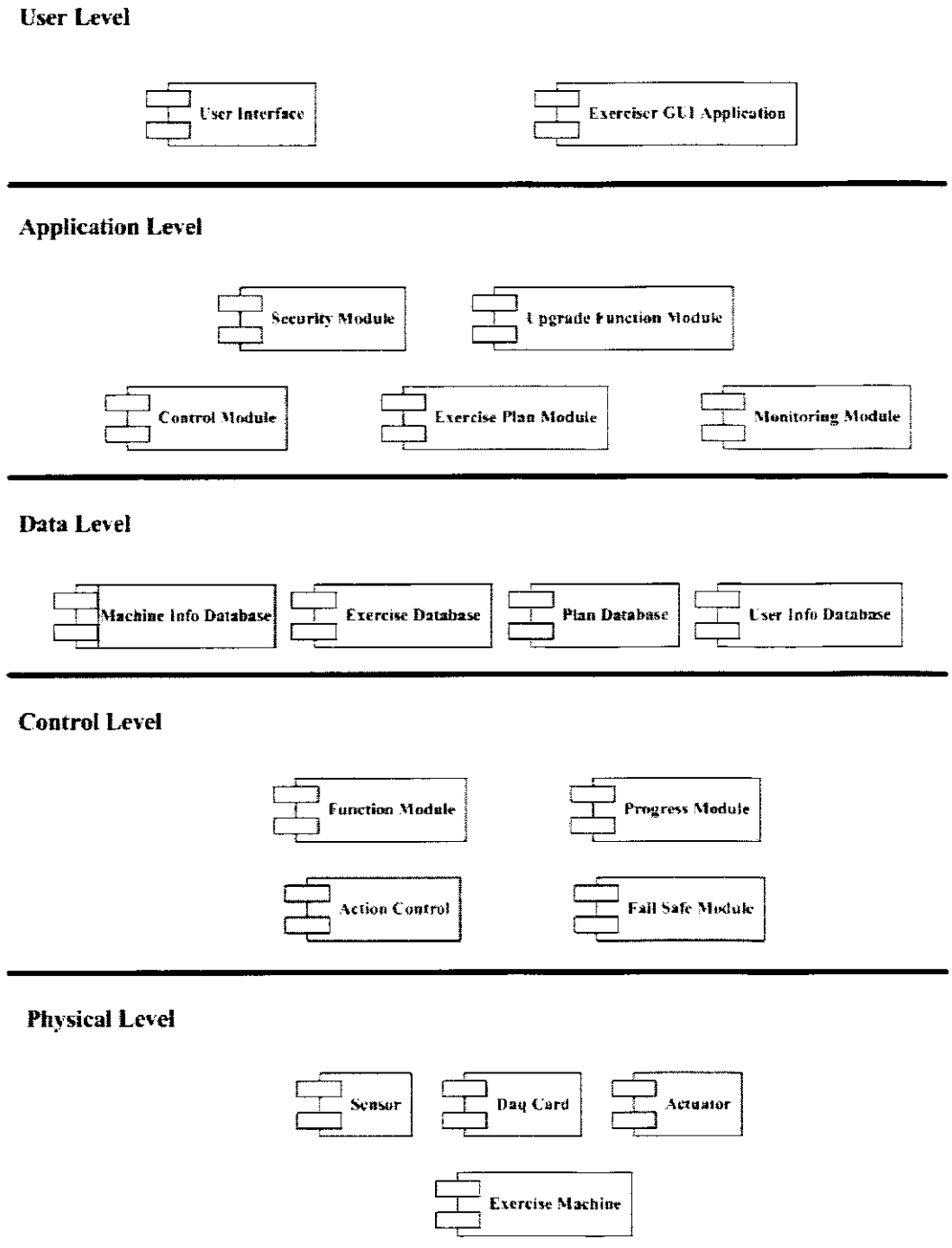


Fig 4.33 The lever structure

4.5 Test

4.5.1 Test of database

To test whole system, a test press sensor and a test database were installed. Press sensor was installed to test the control system by engineer who developed control system, and database was installed to test the information system by information system developer.

Compared with information system database and the test database, there was no difference, although the data was stored in different database.

4.5.2 Test of functionality

Functionality of project

The functionality of the developed test module included the follows:

Exerciser

- Exercise on exercise machine
- Monitor exercise state local
- Catch exercise data
- Make exercise plan

Coacher

- Monitor exercise state local or remote
- Catch exercise data
- Control exercise action locally or remotely
- Make exercise plan

System Admin

- User administration
- Machine administration
- Collect exercise data

System engineer

Upgrade function module

Achieve

All functions above were developed and tested, the relation of class to complete a specified the function is shown below.

For complete exercising, local monitoring and local controlling exercise action, the relation of class is shown in Fig 4.22.

To catch exercise data by exerciser and coacher, the relation of class is shown in Fig 4.34.

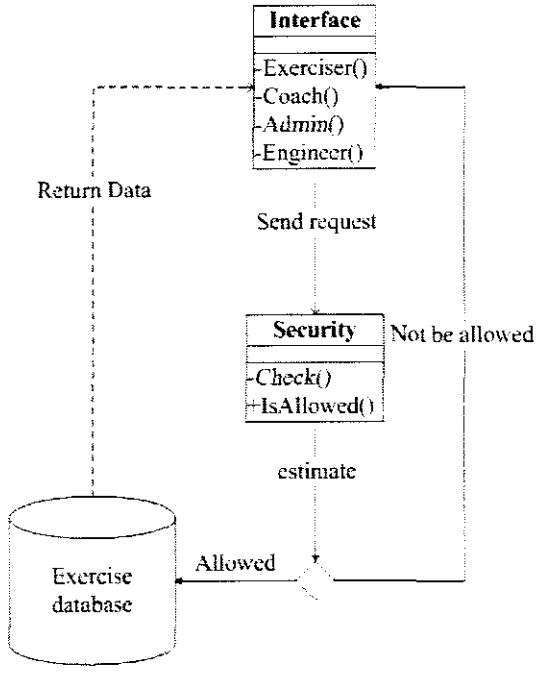


Fig4.34 The relation of class to catch exercise data

For making an exercise plan by exerciser and coacher, the relation of class is shown in

Fig 4.35.

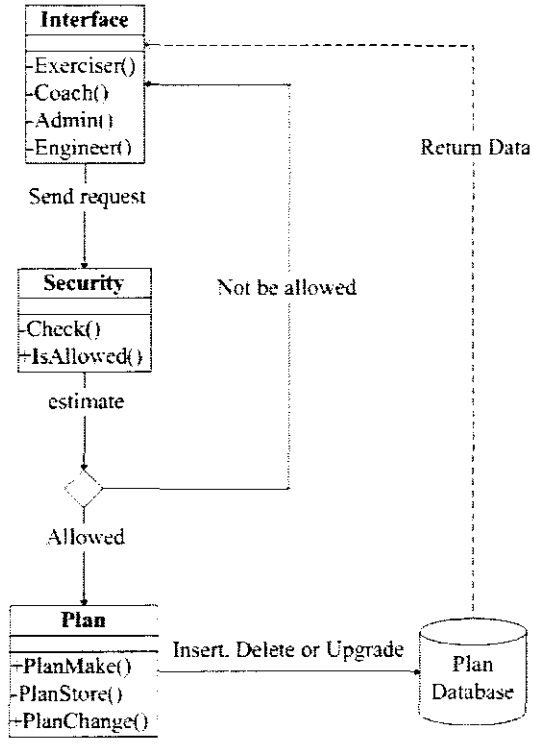


Fig4.35 The relation of class to make an exercise plan

For remote monitoring by coacher, the relation of class is shown in Fig 4.36.

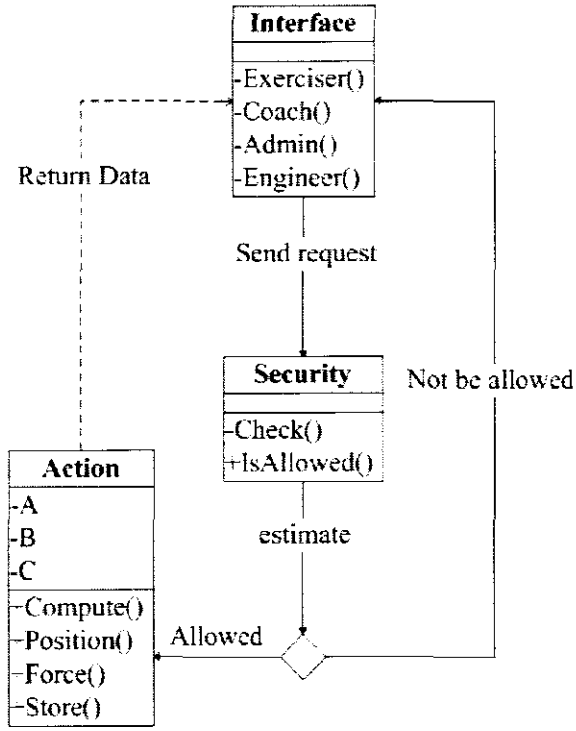


Fig4.36 The relation of class to remote monitoring

For remote controlling by coacher, the relation of class is shown in Fig 4.37.

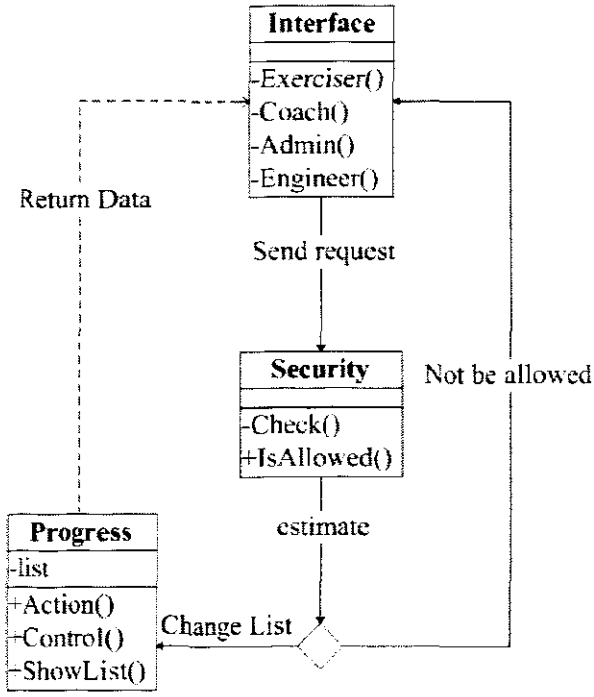


Fig4.37 The relation of class to remote controlling

For user administrating by system administrator, the relation of class is shown in Fig 4.38.

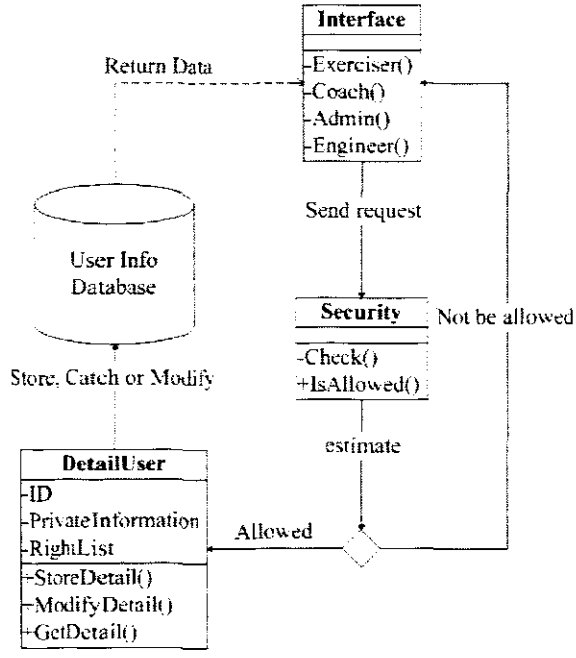


Fig4.38 The relation of class to user admin

For machine administrating by system administrator, the relation of class is shown in Fig 4.39.

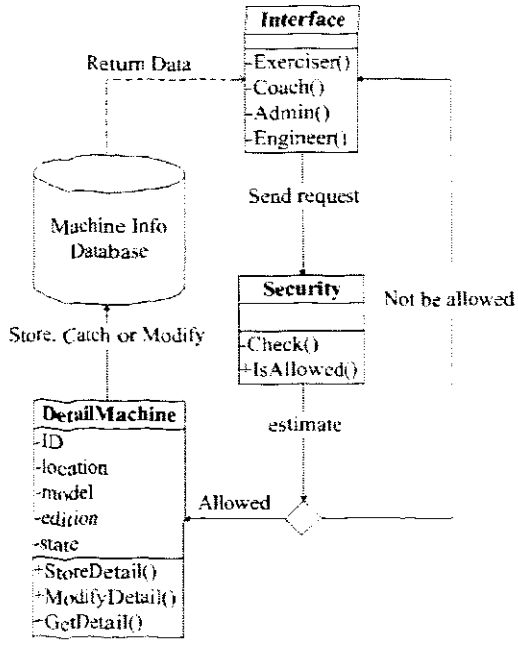


Fig4.39 The relation of class to machine admin

For data collecting by system administrator, the relation of class is shown in Fig 4.40.

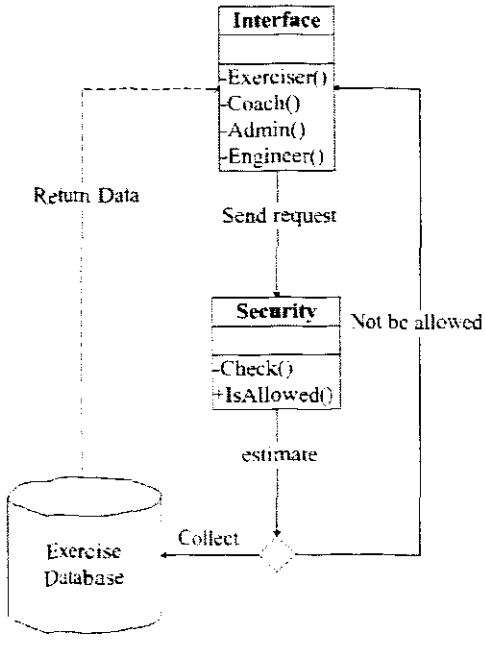


Fig4.40 The relation of class to data collects

To upgrade function module of system engineer, the relation of class is shown in Fig 4.41.

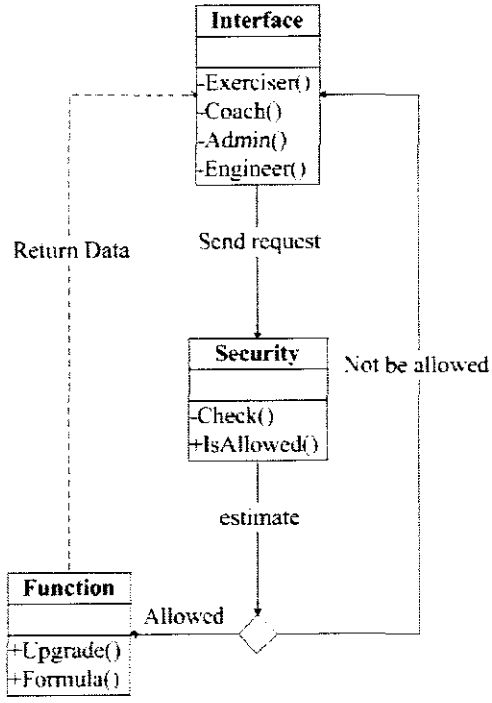


Fig4.41 The relation of class to upgrade function module

4.6 Evaluation

The information system architecture had some incomplete functionality that has been discussed above because of the limitation of time and self-directed ability. In this section, a proposal was set forward for an advanced feature that was the result of in-depth research in information system field. It was hoped that it would be useful in the next step in developing the architecture.

Functionality for a gym

Make exercise plan automatically

Measure rhythm of the heart

A payment module

Combine with gyms

Exercising in different gyms using the same data for exerciser

Share the data between different gyms

4.7 Conclusion

In this chapter, case study, a greater Electronic Fitness Management System was introduced, environment and implementation technology were provided. This analysis impacted greatly on design and implementation of the information system architecture. It provided the descriptions of this Architecture, and supported the use-case to model the architecture.

CHAPTER 5 CONCLUSIONS AND RECOMMENDATIONS

This section contained the conclusions and recommendations. Six questions were answered: What was the primary purpose of this research project? What objectives have been achieved or not completely achieved in this project? Have the functionalities been achieved as per design? What specific technology frameworks have been used to develop this project? What lessons were learned in the project management of this project? The answers concluded this dissertation. Then special and meaningful recommendations were made to the partners and potential users in industry and commerce, followed by recommendations for further research.

5.1 CONCLUSIONS

5.1.1 The primary purpose of this research project

The primary purpose of this research project was to develop a novel dynamic resistance exercise machine with onboard data management. This new technology optimizes exercise effectiveness for users. Our exercisers will have advanced diagnostic capacity, and will capture user data that can be manipulated into meaningful information for use by Doctors and Coachers.

This project established a full integrated technology that combines the following: dynamic resistance exercising, and the electronic capturing / managing of data drawn from the machine user. This required a paradigm shift from single function exercise equipment to intelligent exercise equipment integrated within an IT based information network.

In this project, Information System's driven, and pneumatically actuated exercise equipment was a forward-facing unit. Onboard was the Data Exercise Management

System known as DEMS. The DEMS communicated exercise and fitness data, via an Intelligence Trading House, to Doctors and Coachers. This project novelty lies in the exercises configured for individuals, which allows for seamless exercise routine, without the need for physically adjusting 'weights'. DEMS is an additional novelty offering useful empirical information on exercise status to interested third parties.

5.1.2 The objectives achieved

The following goals and objectives have been set out and achieved in this project:

Objective 1.1: The exerciser stored his data of exercise in database system;

Objective 1.2: Coacher captured data from database to analysis for the exerciser;

Objective 1.3: Coacher monitored and supervised exercise real time;

Objective 1.4: Coacher made an exercise plan for next time;

Objective 1.5: The exercise machine set default exercise mode and intensity for a new exerciser;

Objective 1.6: The exercise machine set exercise mode and intensity follow the plan that coacher was made;

Objective 1.7: The exercise machine set exercise mode and intensity by existent data of exerciser;

Objective 1.8: The exercise machine set exercise mode and intensity manually.

These objectives were met / not completely achieved:

Objective 1.5: The exercise machine set default exercise mode and intensity for a new exerciser;

Objective 1.7: The exercise machine set exercise mode and intensity by existent data of exerciser;

5.1.3 The functionalities that have been achieved as per design

This project achieved its basic function described as follows:

The exerciser can exercise on the machine, set exercise mode and intensity, monitor state of exercise, catch exercise data from database and make exercise plan himself.

The coacher can monitor state of exercise local or remote; control exercise machine locally or remotely; catch exercise data of his exerciser and make exercise plan for his exerciser.

The system administrator can manage user information, including add, delete and modify the information of exerciser, coacher, system administrator and system engineer; manage machine information and collect the data of exercisers.

The system engineer can upgrade function module.

5.1.4 The specific technology frameworks that have been used to develop this project

This project was based on teamwork; therefore, descriptions of every module were needed, which all members could do. The Unified Modelling Language was a third-generation method for specifying, visualizing, and documenting the artefacts of an object-oriented system under development. It was useful for

1. modelling systems (and not just software) using object-oriented concepts
2. establishing an explicit coupling to conceptual as well as executable artefacts
3. addressing the issues of scale inherent in complex, mission-critical systems
4. creating a method usable by both humans and machines

Object-oriented programming in .NET platform was used. This project application framework was based on a close approximation of physical, control, application, database and user interface design pattern. This enable programmer focused on his own field only, not necessary to worry about how to complete other modules and the manager of this project also employed more programmers to develop at same time, in the other words he can submit the whole project ahead of schedule by using same cost (people × month).

5.1.5 Lessons that were learned in the project management of this project

Based on previous detailed analysis of project methodologies it was seen that the best methodology to use can be changed, depending on the particular project, the situation and objectives. Some situations could require a hybrid approach. Especially with regard to providing more detailed and exact estimates of long-term energy efficiency improvement potentials, technological learning over time and related costs will require an in-depth analysis and partly innovative methodological approaches.

5.2 Recommendations

It is recommended to this project in terms of the information system architecture that they:

1. petition the project development team to review and comment upon the information system architecture.
2. examine and fix the specific bugs.
3. furthermore, develop and perfect functionality and interface between different gyms.

Against the background of the thematic issues discussed, there are more recommendations, namely:

1. that more support of communities for share instrument and information between different gyms should be encouraged;
2. that the global health and fitness industry should support and encourage free internet access for collect data of exercisers;
3. that a stable financial support of health and fitness industry for share instrument and information should be ensured;
4. that a more efficient project methodology should be created to develop software engineering.

REFERENCE

.NET Framework Overview Microsoft Corporation Retrieved: Oct 15 2005, from http://research.microsoft.com/collaboration/university/europe/Events/AcademicDays/UK-IE/2003/An_overview_of_.NET.ppt

Agrawal R. (1991) Static Type Checking of Multi-Methods.

Albus J. (1991) Outline for a Theory of Intelligence, *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 21, No. 3, May/June, pp.473-509.

Albus J. S. & Meystel A. M. (1997) Behavior Generation in Intelligent Systems, NISTIR No. 6083, NIST, Gaithersburg, MD.

Answers.com Retrieved: Oct 16 2005, from http://www.answers.com/topic/unified-modeling-language

Astrom K. & Häglund T. (1999) PID Controllers Theory, Design and Tuning. *Library of Congress Cataloguing-in-Publication Data.*

Barker I. n.d. What is information architecture? Retrieved: Oct 02 2005, from http://www.steptwo.com.au/papers/kmc_whatisinfoarch/

Biography.ms Retrieved: Oct 15 2005, from http://unified-modeling-language.biography.ms/

Black A. (1986) Object-Structure in the Emerald System. *OOPSLA '86 Conference Proceedings, SIGPLAN Notices (Special Issue)*, Vol. 21, n0. 11, pp 78-86.

Black A., Hutchinson N., Jul E., Levyand H. & Carter L. (1987). Distribution and Abstract Types in Emerald. *IEEE Transactions on Software Engineering*, Vol. SE13, no. 1 Jam., pp 65-76.

Booch G. (1987) Software Components With Ada, Structures, Tools, and Subsystems. *Benjamin Cummings.*

Booch G. (1991) Object-Oriented Design With Applications. *Benjamin Cummings.*

Booch G. (1994) Object-Oriented Analysis And Design With Applications, 2nd Ed. *Benjamin Cummings.*

Brainboost n.d. what is information architecture. Retrieved: Oct 12 2005, from http://www.brainboost.com/search.asp?Q=what+is+information+architecture&Submit=Ask

Bytheway A. (2002) The Logic Model.

Cai Z. (1997) Intelligent Control: Principles, Techniques, and Applications World Scientific.

Cardelli L. & Wegner P. (1985) On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys* vol. 17.

Chambers C. (1992) The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages. *Dept of Computer Science, Stanford University, March.*

Chambers C. (1993) Predicate Classes. Proceedings ECOOP '93 O. Nierstrasz,

LNCS 707. *Springer-Verlag. Kaiserslautern. Germany* July pp 268-296.

Coad P. & Yourdon E. (1991) Object-Oriented Analysis, 2nd ed. *Englewood Cliffs. NJ. Prentice Hall.*

Coplien J. O. (1992) Advanced C++ Programming Styles and Idioms. *Addison Wesley.*

Cox B. J. (1991) Object-Oriented Programming, An Evolutionary Approach. *Addison Wesley.*

Erlbacher E. A. (1995) Force Control Basics.

Ferraresi C., Giraudo P. & Quaglia G. (1994) Non Conventional Adaptive Control of a Servopneumatic Unit for Vertical Load Positioning. *Proceedings of the 46th National Conference on Fluid Power.*

Ferraresi C., Raparelli T. & Velardocchia M. (1990) Studio della Stabilità e della Prontezza di Sistemi di Posizionamento Pneumatici. *X Congresso Nazionale dell'Associazione Italiana de meccatronica.*

Gao X. & Feng Z. (2004) Design study of an adaptive Fuzzy-PD controller for pneumatic servo system.

Gao Y. & Yuen C. K. (1993) A Survey of Implementations of Parallel, Concurrent, and Distributed Smalltalk. *ACM SIGPLAN Notices.* Vol 28, No. 9, Sept.

HKSAR (2005) OOM Procedures Manual. Retrieved: Oct 21 2005, from http://www.ogcio.gov.hk/eng/prodev/eg52.htm

Hathaway B. (1995) Object-Orientation FAQ.

Hesselroth T., Sarkar K., van der Smagt P. P., & Schulten K. (1994) Neural Network Control of a Pneumatic Robot Arm, *IEEE Trans. on Systems, Man, and Cybernetics*, Vol. 24, No. 1, Jan. pp.28-38.

Jacobson I. (1992) Object-Oriented Software Engineering - A Use Case Driven Approach. *ACM Press/Addison Wesley.*

Jones R. (1992) Extended type checking in Eiffel. *Journal of Object-Oriented Programming*, May issue, pp.59-62.

Kim W. & Lochovsky .F. (1989) Object-Oriented Concepts, Applications, and Databases.

Kimen S. (2003) 10 questions about information architecture. Retrieved: Oct 14 2005, from http://builder.com.com/5100-31-5074224.html

Klein A. (1993) Employment of fuzzy Logic for Controller Adaption in fluid Power Cylinder Actuators. *European Journal of Fluid Power, Noc.*

Lakoff G. (1987) Women, Fire, and Dangerous Things: What Categories Reveal About The Mind. *UOC Press.*

LaLonde W. R. & Pugh J.R. (1990) Inside Smalltalk: Volume 1. Prentice Hall.

Lemmon M. D. (1994) On Intelligence and Learning, report of the Task Force on Intelligent Control, *IEEE Control Systems*, 0272-1708/94, June, pp. 63.

Ling N. (1993) A Simple Expert System for the Reasoning and Systolic Designs,

IEEE Comp. Soc. Press, CCC: o 8186 3492 8/93, pp.128-131.

Liskov B. & Wing J. M. (1993) Specifications and Their use in Defining Subtypes. OOPSLA 93, pp 16-28. *ASM SIGPLAN Notices*, V 28, No 10, Oct. 1993. A-W ISBN 0-201-58895-1.

Madsen O. L., Pedersen B. M. & Nygaard V. (1993) Object-oriented programming in the BETA programming language. *Addison-Wesley.*

Martin J. & Odell J. (1992) Object-Oriented Analysis and Design, Prentice-Hall, Englewood Cliffs, NJ.

Maurer D. Retrieved: Oct 01 2005, from http://www.steptwo.com.au/papers/kmc_whatisusability/index.html

Maybeck P. S. (1979) The Kalman Filter: An Introduction to Concepts. *Stoch. Models, Estim. and Control*, Vol. 1, 1979, pp. 3-16.

Meyer B. (1988) Object-Oriented Software Construction. *Prentice Hall.*

Meystel A. and Messina E. (2000) THE CHALLENGE OF INTELLIGENT SYSTEMS *15th IEEE International Symposium on Intelligent Control (ISIC 2000) Rio, Patras*, GREECE 17-19 July.

Microsoft .NET Framework Reviewers Guide. Microsoft Corporation Retrieved: Oct 03 2005, from http://download.microsoft.com/download/VisualStudioNET/Utility/7.0/W9X2K/EN-US/frameworkevalguide.doc

Microsoft Retrieved: Oct 04 2005, from

http://www.microsoft.com/presspass/press/2000/Jun00/ForumUmbrellaPR.mspx

Ming-Chang S. & Tsung-Wei L. (1998) On-line Learning Neural Fuzzy Control the Position of a Pneumatic Cylinder. *International Conference on Advanced Mechatronics.*

Montlick T. (1999) What is Object-Oriented Software?

Moreno L. H. (2000) CONTROL OF A PNEUMATIC SERVOSYSTEM USING FUZZY LOGIC.

MSDN training Retrieved: Oct 23 2005, from http://www.microsoft.com/

Musto J. C. & Saridis G. N. (1998) A Reliability-Based Formulation for Intelligent Control International. *Journal of Intelligent Control and Systems*, Vol.2, No. 2, pp. 193-209.

National Instruments a. Retrieved: Oct 21 2005, from http://www.ni.com/labviewse/ltb.htm

National Instruments b. Retrieved: Oct 24 2005, from http://www.ni.com/devzone/reference/books/

OMG 2002 Retrieved: Oct 15 2005, from http://foldoc.doc.ic.ac.uk/foldoc/foldoc.cgi?UML

Pang G. K. H. (1991)A Framework for Intelligent Control. *Journal of Intelligent and Robotic Systems*, No. 4, pp. 109-127.

Poggio N. n.d. Theory: The design of experiences. Retrieved: Oct 05 2005, from

http://www.ices.utexas.edu/~natacha/CATTt/theory.html

Polycarpou M. n.d. Technical Committee on Intelligent Control. Retrieved: Oct 03 2005, from http://www.ieeecss.org/TAB/Technical/TCIC/

Public Life n.d. Retrieved: Oct 15 2005, from http://publiclife.co.uk/glossary.html

Raj R. K. & Levy H. M. (1989) A Compositional Model for Software Reuse. *The Computer Journal,* Vol 32, No. 4, 1989.

Rational Software Retrieved: Oct 17 2005, from http://www.microgold.com/version3/faq_provided_by_rational_softwar.htm

Rumbaugh J. (1991) Object-Oriented Modeling and Design. *Prentice-Hall.*

Sciore E. (1989) Object Specialization. *ACM Transactions on Information Systems.* Vol. 7, No. 2, April , p 103.

Shlaer S. & Mellor S.J (1988) Object-Oriented Systems Analysis: Modeling the World in Data.

Simson L., Garfinkel R. & Michael K. (1993) NeXTSTEP PROGRAMMING STEP ONE: Object-Oriented Applications. *Springer-Verlag.*

Sloman A. (1978) The Computer Revolution in Philosophy: Philosophy Science and Models of Mind Hassocks: Harvester Press

Sloman A. (1994) Explorations in Design Space, proceedings. *11th European Conference on AI Amsterdam.*

Song Y. D. & Mitchell T. L. (1993) A new and Simple Control Strategy for Multiple Robots Involving Redundant Motion, *IEEE Proceed. of the 32nd CDC. San Antonio. TX*, Dec. pp. 1124-1125.

Sorli M., Gastaldi L., Codina E., & De las Heras S. (1999) Dynamic Analysis of Pneumatic Actuators. *Journal SIMPRA.*

Stroustrup E. A. (1990) The Annotated C++ Reference Manual. *Addison Wesley.*

Stroustrup E. A. (1991) The C++ Programming Language (2nd edition).

Ungar D. & Smith R. B. (1987) The Self Papers.

Using Graphical Programming Throughout the Development Cycle with NI LabVIEW (2005) In Automated test summit 2005.

Vecchiola C., Gozzi A., Coccoli M. &Boccalatte A. (2002) An Agent Oriented Programming Language Targeting the Microsoft Common Language Runtime.

White D. A. & Sofge D. A. (1993) Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches Van Nostrand Reinhold, NY.

Zimmermann H. J. (1991) Fuzzy Set Theory and its Aplications second edition. Kluwer Academic Publishers.