

https://dspace.org/

Object oriented development a logical approach to control system software design

Bricker, R I

2016-08-26
http://hdl.handle.net/10539/20943

Downloaded from a DSpace repository. https://dspace.org/

OBJECT ORIENTED DEVELOPMENT, A LOGICAL APPROACH TO CONTROL SYSTEM SOFTWARE DESIGN

R.I. BRICKER

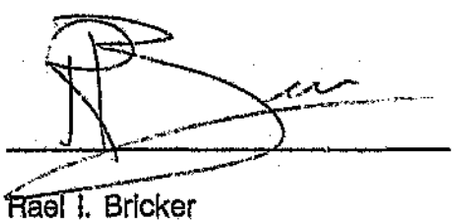
STUDENT NO 82/1992/3

A dissertation submitted to the Faculty of Engineering, University of the Witwatersand Johannesburg, in fulfillment of the requirements of the degree of Master of Science in Engineering.

Johannesburg April 16, 1990

DECLARATION

I declare that this dissertation is my own unaided work. It is being submitted for the degree of Master of Science in Engineering in the University of the Witwatersand, Johannesburg. It has not been submitted before for any other degree or examination in any other University.



Rael I. Bricker

17th day of APRIL 1990

ABSTRACT

Automation is currently being used to an ever increasing degree in industrial plants. However most of these applications require only a few basic control concepts repeated for each piece of equipment. There is often a fair amount of interaction between pieces of equipment in terms of safety interlocking and sequencing. Despite this each piece of software remains an autonomous entity receiving the necessary external data it requires. The principles of OBJECT ORIENTED DESIGN are suited to the development of this type of software. This dissertation will demonstrate that object oriented development has distinct benefits over more classical design techniques. Generalized software for controlling a minerals processing plant will be conceptually designed, using techniques implemented in C++ to demonstrate the basic hypothesis. This will attempt to encompass all the available design techniques of object oriented design into an area that has traditionally developed its own software development paradigm.

DEDICATION

This dissertation is dedication to the special people in my life who gave me the support and encouragement to continue my studies.

To my parents and sister who have hardly seen me in two years due to my studies, and to my wife Rafaella (Rafi) who gave up numerous weekends and nights whilst I was studying.

ACKNOWLEDGMENTS

I would like to thank Professor A. J. Walker for his support and encouragement in the times when I became despondent, and worried about being a **"padvinder".**

I would like to thank the Faculties of Engineering and Business Administration for allowing me to register for two degrees simultaneously.

Thanks must go to Kenwalt who procured a copy of Zortech C++ which allowed me to complete this dissertation and taught me so much about the practicality of control system technology.

Lastly, I must thank the staff at Anglo American Control and Instrumentation Department who started my career in the process control field and provided me with the fundamental ideas that created the hypothesis of this dissertation.

TABLE OF CONTENTS

DECLARATION i

ABSTRACT ii

DEDICATION iii

ACKNOWLEDGMENTS iv

TABLE OF CONTENTS v

1.0 INTRODUCTION 1

1.1 Programmable Logic Controllers 1

1.2 Object Oriented Design and Programming 5

1.3 Basic Hypothesis 8

1.4 Conclusion 12

2.0 CURRENT PLC LANGUAGE TRENDS 14

2.1 Introduction 14

2.2 PLC Languages 15

2.2.1 Ladder Logic 15

2.2.2 Control system Flowsheet 20

2.2.3 Boolean Logic 23

2.2.4 Sequential Function Chart 24

2.3 Conclusion 30

3.0 OBJECT ORIENTED SOFTWARE DEVELOPMENT TECHNIQUES 31

3.1 Data Abstraction 32

3.2 Information Hiding 33
3.3 Dynamic Binding. 34
3.4 Inheritance 36
3.5 Design methodology. 37
3.6 Conclusion 41

4.0 CONTROL SYSTEM SOFTWARE DESIGN 42
4.1 Introduction 42
4.2 Generalized Software Design 44
4.3 Conclusion 49

5.0 SIMPLE OBJECT ORIENTED IMPLEMENTATION 50
5.1 Introduction 50
5.2 Physical Input and Output 51
5.2 General Drive Class 56
5.3 Additional Derived Classes 63
5.3.1 Drive with no features 63
5.3.2 Drive with start siren 64
5.3.3 Drive with Safety Timer 66
5.4 Conclusion 69

6.0 ADDITIONAL INFRASTRUCTURAL SOFTWARE 70
6.1 Introduction 70
6.2 Data Management 72
6.3 Interlocks 78
6.4 Sequences 82
6.5 Menu Management 88
6.6 File Storage and Retrieval 90
6.7 Conclusion 95

7.0 SOFTWARE FOR CRUSHING AND SCREENING PLANT 96
7.1 Introduction 96
7.2 Functional Description 96
7.3 Start-up Sequence 97
7.4 Interlocking 99
7.5 Software design 101
7.6 Conclusion 104

8.0 CONCLUSION 106

REFERENCES AND BIBLIOGRAPHY 112

APPENDIX 1 I

1.0 INTRODUCTION

"Programmable controllers are being applied in ways never envisioned by the equipments original developers, who were primarily concerned with the task of relay replacement"

Ken Ball, Consulting Editor
Walter J. Maczka, Editor in Chief
Intech July 1987

"Object Oriented Programming with its claims of improved programmer productivity and easy program maintenance, is emerging as a vital force in the programming community. This despite the fact that object oriented languages were not generally available until fairly recently."

Eva White, Technical Editor
Rich Malloy, Technical Editor
Byte August 1986

1.1 Programmable Logic Controllers

Programmable logic controllers (PLC) began their emergence in the early 1970's due to the need for an accelerated changeover cycle in the motor industry in the USA. (Nolett 1986 p58) PLCs originated as a replacement for relay panels. Thus, the first language developed for PLC systems was called ladder or relay logic.

PLC languages have for a long time kept the emphasis on software that is easily maintainable by plant personnel that do not have a formal computer training. In addition, PLC languages have tended to be mainly graphical and three of the four major languages (discussed in chapter two) are graphically based. It is for this reason that **"software driven control systems have been programmed by**

control engineers rather than software engineers."(Waine 1987 p63) This is however changing at quite a rapid pace as the large PLC has become a small industrial computer and the **" 11 pin relay base has become a 20 word software construct."** (Waine 1987 p63)

One of the major problems faced by software engineers designing PLC software is that the disciplines of the last 20 years of software engineering have not followed through into the PLC field. The major reason for this has been the hardware constraints, the lack of programming languages and the heavy emphasis on easy to use graphical languages. Both have made rapid advancements in the last five years and the emphasis is definitely towards software engineering and not control engineering.

One aspect that has evolved over the last few years is the development of the distributed control system (DCS). This has evolved from the use of multiple independent control systems, perhaps PLC based, linked to a central operator station. The issue, however, is that DCS suppliers no longer **"program"** the DCS, they tend to use the term **"configuration"**. The configuration of a DCS (such as Hartmann and Braun Contronic P and Foxboro I/A) uses an interactive configuration program. Thus, when configuring a DCS, a programmer is prompted for information in an English style format, as opposed to a programming language. This does not imply that there is no programming required, but rather the converse. The programming is done at a much higher level, to interpret the configuration program and convert that into a form that will

generate the control system. This is providing a completely new light on the control system design and development arena. The control or software engineer, developing the control system, is no longer stuck behind a terminal, coding in a traditional language.

What has evolved is that the programmer now develops a control philosophy, and converts that into configuration sheets, that can be typed into the control system by a clerical worker. This does not imply that development of control systems has become easier, but rather that the tedious task of programming has become simplified. If the design of the control philosophy is wrong, the control system will still not work.

The implication of the developments in the DCS environment, indicate this definite shift away from the need for graphical based PLC languages, to a higher level form of system configuration. This will perhaps speed up the development of high level languages for control system development.

A distinction must be made between the user programming and the system programming, as is the case in any computer system. Traditionally, PLC programming has been a mixture of both areas. The programmer would be presented with a programming environment, in which to develop all the control constructs and then link these together to form a coherent program. In certain of the programming environments, the programmer does have the facility to create functions and pseudo-procedures. (e.g. Siemens PLCs) This facility does

allow for the setup of an infrastructure and then the development of the main program that calls these functions.

However, the languages used are still at a very low level. The situation in the DCS environment is that the development of the standard functions is carried out by the control programmer, and then the configuration is carried out at a higher level than with PLC systems. However, the basic functions remain part of the **"user program"** as opposed to the system environment. The principles of control and control philosophies will be discussed throughout this dissertation, however it must be noted that there are as many philosophies about control systems as there are control engineers. Thus it would be impractical to have a PLC or DCS system that has every control system philosophy built into its system environment.

However, the program generated as part of the research for this dissertation, will demonstrate that the system environment can be easily adapted for each control situation. In addition, it will show that this adaptation and system setup is easily achieved using the techniques of object oriented software design.

1.2 Object Oriented Design and Programming

Object Oriented design and programming has emerged in the 1980's as a superior method of software design. (White and Malloy 1986 p137) The emphasis, however has been the need for higher levels of programming language to cope with the degree of abstraction required.

The term Object-Oriented programming evolved as a description of the Smalltalk programming environment as developed at Xerox PARC. The major difference between conventional programming (procedure-oriented) languages and so-called Object-oriented languages lies in the concept of data procedures vs object messages (Pascoe 1986).

In the more conventional languages active procedures act on passive data that is passed to them. An example of this would be the procedure to find the square root of a number : sqrt(x). In data procedure languages the procedure would take the number and return the value of the square root. Object-oriented languages on the other hand employ an object ordered approach. In this case the data is not passed to the sqrt procedure, rather the message sqrt is passed to the data which will perform the function on itself. It is for these, amongst other reasons,that programmers are calling for higher levels of programming language to cope with the abstraction required by an object-oriented approach to programming. The principles of object oriented design will be discussed in detail

in a chapter three, and the benefits will be discussed throughout this dissertation.

It is felt that the emphasis on object oriented design has been on a academic level in terms of requiring higher levels of abstraction in the programming languages rather than applying these principles to practical issues. There have been numerous papers written on the subject of object oriented design that criticize the languages that currently exist. (Duff 1986) Very few however tend to accept what is available and attempt to apply these object oriented design techniques in slightly modified forms to these languages. One of the first languages to evolve that incorporated the principles of Object Oriented design was Smalltalk. The major problem with this was the inherent inefficiency in the language. The " **programs run a lot slower and required a lot of computer resources. Also, you generally needed the entire development environment loaded to run the program.**" (Bright 1988) A number of these problems were addressed by Stroustrup (1987) in developing the C++ language. This used the run time efficiency of a compiler combined with the advantages of object oriented design. Both languages have their merits, with Smalltalk being an excellent prototyping and development language and C++ (and other compiler languages) delivering efficient portable code.

The fundamental principle of object oriented design that is applied throughout this dissertation is that of dynamic binding. Simply stated, a number of objects are defined and each will respond in fundamentally different ways to the same message passed to them. The only commonality between them is that they are

derived from a simple base class. This will expounded upon in later chapters and demonstrated in the C++ programs generated. This fundamental principle of object oriented design allows for the simplification of a number of infrastructural systems as well as the development of a high level programming interface for PLC systems. It is this basic premise that forms the foundation of this dissertation.

1.3 Basic Hypothesis

The basic contention of this dissertation is that these two totally juxtaposed concepts can mesh into one powerful method of developing control system software. This would incorporate the advantages of both concepts whilst attempting to overshadow the weak points of each one. Simply stated, it is the objective of this dissertation to demonstrate that the principles of object oriented programming and software design can be used to effectively design PLC systems and hence their software. In addition this dissertation will show why object oriented design should be used, as opposed to some other software development paradigm.

One of the major constraints of this work is that at present there are no PLC languages that allow for object oriented programming. Therefore the emphasis of this entire dissertation is that, firstly, the concepts of object oriented design can be applied to the development of software for PLC systems. Secondly, the principles of object oriented design can effectively be used to generate a high level control system programming interface that allows for the more efficient use of programming time and hence a saving in monetary terms.

The most fundamental question that must be answered is " **Why object oriented design as opposed to some other software development paradigm?**" The answer must be given in the light of a number of factors. On the most simplistic level, there is a major trend toward object oriented design principles in the

software world. Object oriented design, provides a number of benefits that are not available through using more conventional development paradigms. The major benefit, that will be demonstrated is the use of the classing and subclassing (inheritance) facility within the object oriented environment. This allows for a fairly simple definition structure to be used as a starting point for any PLC program. As the complexity increases the base classes of equipment within the plant being controlled, can be subclassed and levels of complexity added.

The second major benefit is arises from the use of the information hiding aspect of object oriented design. This allows the interface connections to the physical input and output to only be available to a specific piece of equipment. It does not become globally available as is the case in PLC systems. This decreases the chance of errors occurring when changes to the control algorithm tries to access inputs or outputs associated with another piece of equipment.

An area that was discussed previously is that of dynamic binding. This almost eliminated the need for type checking within a program as the system decides at runtime, what actions are need in response to a particular message being passed to a specific object. Strong type checking is done at compile time, however, it will be shown that in the context of this dissertation, the dynamic binding forms the heart of the code.

Another major benefit is that is believed that the program code generated may use substantially less program memory than current PLC programs. The reason

for this saving lies in the fact the different types of equipment are globally defined within the system environment, and the only additional code is their definition section. Conventional PLC programs require that often identical pieces of code be copied and only the definitions changed. This requires a substantial overhead in terms of memory usage. This is an extremely difficult figure to quantify due to the lack of object oriented control system programming environments. However, if this does prove to be a major benefit in the future, it will mean a reduction in hardware costs.

A similar benefit to the one discussed above is that of software reusability. This has been discussed by Meyer (1987) and in the context of PLC systems will allow for the more efficient use of programmers.

This area however specifically excludes the PLC environment, however, it should be emphasized that PLCs are slowly moving away from the classical low level language in the direction of languages such as C, Pascal and Forth. (Programmable Controller Product Survey 1987) This will allow for the object oriented techniques to be applied to an even greater extent.

It must also be realized that until the control system languages fully incorporate the principles of object oriented design, it will not be possible for programmers to make use of all the features of object oriented design. However, another major proposal of this dissertation is that the principles of object oriented design are fundamentally sound and provide numerous benefits as compared to other

software development paradigms. The implication, is that developers of control system software, begin using the techniques that will be discussed, in a more structured fashion until the control system languages have caught up with the design techniques.

1.4 Conclusion

The original basis for this dissertation arose when it was discovered that PLC programmers, were, in an informal manner, applying the principles of object oriented development to their design. The dissertation has evolved to a much higher level of programming environment, whilst still maintaining the most basic premises of control system software design. The conclusion is that the techniques of object oriented design will provide a sound basis for the future development of control system software. This basis will revolve around the development of the system infrastructure as well as the ease of development in the user program arena. Finally, it will also present a strong case for the use of object oriented development as an alternative to the more conventional software development paradigms. This will also refute the calls for higher and higher levels of abstraction in the languages, and show that the languages that exist, perhaps not in the PLC environment, do encompass the basic principles of object oriented design.

Chapter two considers the current state of the art in PLC languages and possible future trends in this regard. Chapter three covers the concept of object oriented design as applied to the design of generalized software. Chapter four deals with the methods in which object oriented design can be applied to the design of control system software. Chapter five considers the implementation of a number of basic types of drive classes in C++ and the implications of using an object oriented language for this purpose. Chapter six describes the infrastructural code

necessary to implement the simulation of a plant, the control system and the supervisory controls within such a plant. The conceptual design of the software for the crushing and screening section of a diamond processing plant is covered in chapter seven. This is an attempt at showing how the principles discussed in the previous chapters may be applied in a real live plant. One of the major constraints is that there are currently no PLC's that are capable of being programmed in an object oriented language. Chapter eight is the conclusion and considers the future of PLCs and Object oriented design.

Appendix 1 contains a full compiled listing of all the software written for this dissertation.

2.0 CURRENT PLC LANGUAGE TRENDS

2.1 Introduction

Programmable logic controllers have since their introduction in the early 1970's proliferated and 1986 sales in Europe were estimated to be in the region of 530 million pounds sterling. (Readman 1987) The original application of the PLC was as a replacement for relay panels. This brought about the first of the PLC languages (Ladder Logic) which was a visual representation of the logic of the relay panel. PLCs are now available in varying sizes and degrees of complexity. At the bottom end of the scale is the National PL16 (Readman 1987) which offers 8 inputs and 8 outputs. In the middle order there are the Siemens 150U series (Siemens 1986), Telemechanique TSX7 (Telemechanique 1987) amongst others, all offering up to 2048 inputs and outputs. At the top end of the scale is the General Electric series 6 offering 16000 digital I/O. (Readman 1987)

PLC programming languages have evolved into a four main types, with each vendor supporting variations on one of these basic types. These four are Ladder logic or Relay logic, Control system flowsheets, Sequential function chart and Boolean logic. Certain languages may include a combination of the above basic languages types. Each of the four areas will now be discussed and some possible variations or combinations of each shown. It is not possible within the scope of this dissertation to cover all the different languages and therefore only these four basic types will be considered.

2.2 PLC Languages

PLC Languages originated as a conversion from relay panels. It was, and still is possible to convert the logic from relay panels directly into PLC logic. For this reason, many PLC vendors refer to a language called Relay Ladder or Relay Logic.

2.2.1 Ladder Logic

Ladder Logic parallels digital combinational and sequential logic, to a large extent. Each vendor has a unique method handling of I/O within their version of ladder logic. This incorporates the mnemonics used, limitations on the number and type of logic gates and the general constraints of the language. These are too many to be enumerated within the scope of this dissertation. However, the basic format of the language remains unchanged, and the symbols used do not vary to any great extent.

The basic concept of ladder logic is that it is analogous to the concept of 'passing power'. This stems back to the relay panels where, as power was applied to one side of a set of relays in series, an action would occur if the 'power was passed' right through to the last relay.

Ladder logic revolves around two **"gates"** viz n/o] [(normally open) and n/c]/[(normally closed). A n/o gate corresponds to a relay whose contacts are open

i.e. not passing power, when there is no current passing through the relay coil. If the desired state of a gate is one where it is closed (passing power) then a n/o gate will be in its desired state when the digital input to which it corresponds is in the logical 1 or On state. The converse applies for a n/c (normally closed) gate in that its desired state is when the input to which it corresponds is in a logical 0 or Off state.

If, for example, an action should occur when an input changes from a logical 1 to a logical 0 then a normally closed n/c gate will be used. In the logical 1 state this gate will be open and hence not passing power and conversely in the logical 0 state will be closed and pass power to the next gate in the sequence.

Ladder logic in its simplest form consists of manipulating bits of input or output data. Figure 1 shows the logic diagram of a simple pump that is started in the field from a push-button switch. To the operator in the field it appears that the action of pushing the start button directly starts the pump. Actually his start signal is merely a **"request to start"** signal sent to the PLC. The PLC also receives an indication of whether the pump has been field stopped (the stop button has been pressed), tripped (due to a fault) or whether it is currently running. If all these signals are in the correct state then the PLC will send a signal to the pump to start. The start signal is then fed back into the beginning to latch itself. This is necessary as the pump in the field must continually have a signal applied for it to continue running. In addition this provides a safety

feature in that if the PLC power is no longer applied then the pump will automatically stop. (fail to safety).

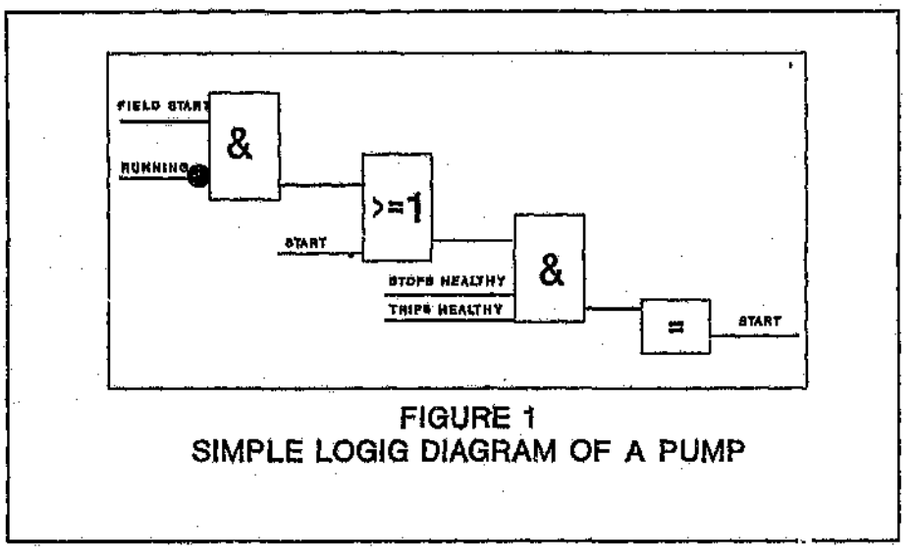


FIGURE 1
SIMPLE LOGIG DIAGRAM OF A PUMP

Figure 2 shows the ladder diagram of the simple field started pump. When the field start button is depressed the corresponding n/o contact will close and assuming that the trips and stops are in a logical 1 state power will be passed to the start contact. This contact (state 1) is also used as a latch around the field start and running contacts as when the field start signal is no longer applied then the start signal must remain on. The running signal is a n/c contact that will open when the pump is running. This is used to prevent any action being taken if the field start button is depressed whilst the pump is running. If, however, the PLC output must be removed, as a safety precaution, after a time interval if the pump has failed to start then a n/c contact will be added as shown i . igure 3. In this example when the PLC output it turned on a timer is started.

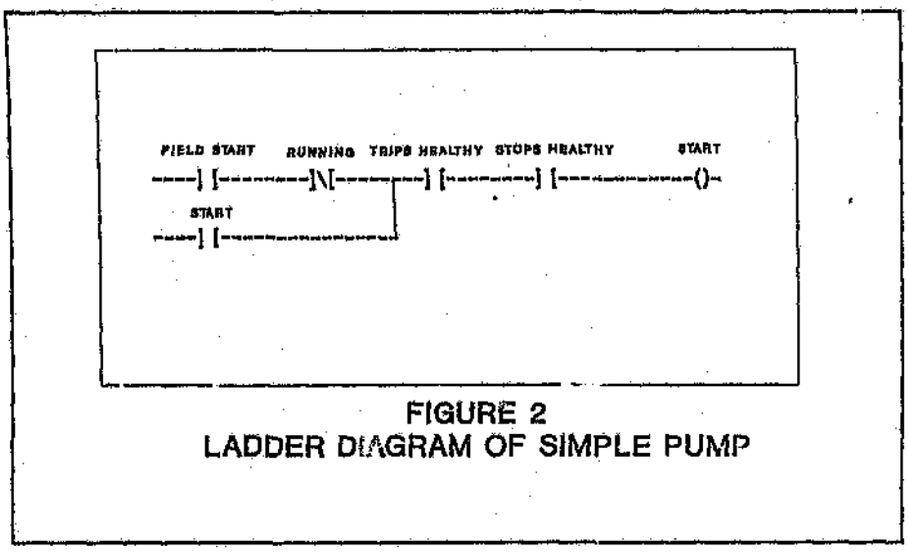


FIGURE 2
LADDER DIAGRAM OF SIMPLE PUMP

The operation of the timer is based on the principle that when the input signal is applied it will begin timing the period that has been set. If it is not switched off (in this case by the running signal) before the end of the time period then it will, at the end of the time period, change the state of its output to a logical 1. This will cause the temp flag (n/c) to become open thus removing the latch on the start signal and causing the start signal to fall away. This is done mainly as a safety precaution so that in the event of a fault the start signal will not constantly be applied. If it were, once the fault had been cleared then the pump would automatically start and perhaps endanger the person who rectified the fault. This is only one of the types of timers available to the programmer, all however depend on the same principle of operation (Siemens 1986). Some however will turn the output on for the timing period or turn the output off on completion of the timing period.

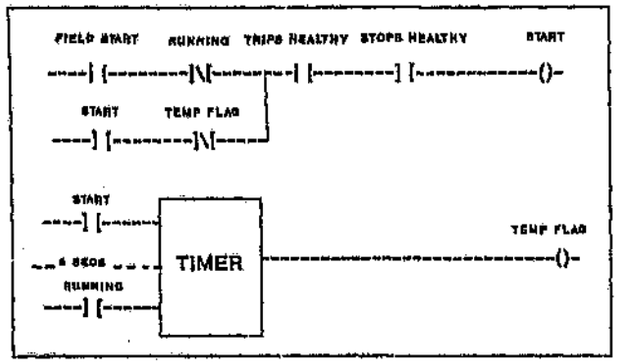


FIGURE 3
LADDER DIAGRAM OF PUMP WITH TIMER

Ladder logic in its simplest form has both advantages and disadvantages. One of the major downfalls is that when using ladder logic as the only form of programming, the entire program is executed during every scan of the PLC. This form of ladder logic does not allow for selective calling of different subroutines and functions in specific instances. A dummy contact can be used to simulate this action, however this will undesirably increase program complexity.

The chief advantage of pure ladder logic is that it is simple to program. This is due to the fact that there are only two possible contacts and contact states. There is no variety of programming statements to be remembered and this decreases the possibility of incorrect typing of statements.

Ladder logic has served as a useful transition from relay panels to the ever expanding field of computerized control. Pure ladder logic is today not being superseded by higher levels of programming but rather forming part of the

subsets of these higher levels languages. Ladder logic will always have a place in the PLC programming area because of its inherent simplicity and ease of use.

2.2.2 Control system Flowsheet

Earlier Figure 1 was referred to as a logic diagram. This form of program representation can be applied directly with certain PLCs. These PLCs use a language called CSF (Control System Flowsheet). This allows the programmer to translate the logic diagrams of the design directly into a machine readable form by typing the flowsheets into the programming unit.

As with ladder logic CSF revolves around two major functions viz the And (&) and Or (>=1) gates. A useful comparison can be found by comparing CSF to Ladder. In Ladder n/o and n/o contacts are put in series or parallel to form "And" and Or functions respectively. If a number of ladder logic contacts are placed in series they will pass power once all conditions have been met thus forming an AND function. If, however, they are placed in parallel when any one condition is met, power will be passed thus forming an OR function. If the gates are inverted i.e. n/o to n/c then the AND will become an OR and visa versa.(Figure 4) With CSF these gates already exist (And,Or) but the inputs to these gates may have to be inverted to take into account any input states. This is shown in figure 5 where the corresponding CSF and ladder logic functions are shown.

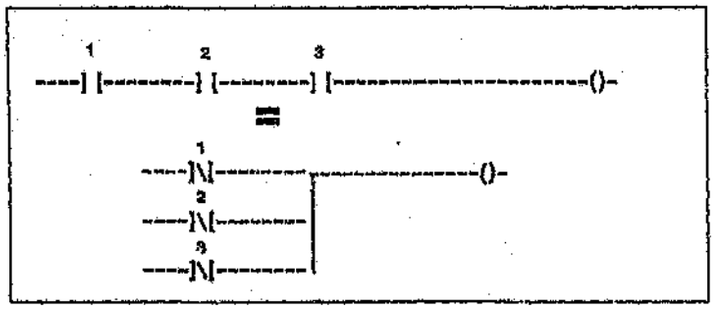


FIGURE 4
COMPARISON OF SERIES AND PARALLEL
LADDER LOGIC

CSF has the obvious advantage that it requires little translation from design to implementation. It also is a breakaway from the translation of relay logic panels. However CSF suffers from the same shortcomings as ladder logic in that the entire program is executed every scan of the PLC, in most implementations. However certain PLCs (Siemens, Telemechanique) do make provision for conditional calling of so-called subroutines. In addition there is the possibility that support personnel will not have the knowledge to maintain programs written in CSF.

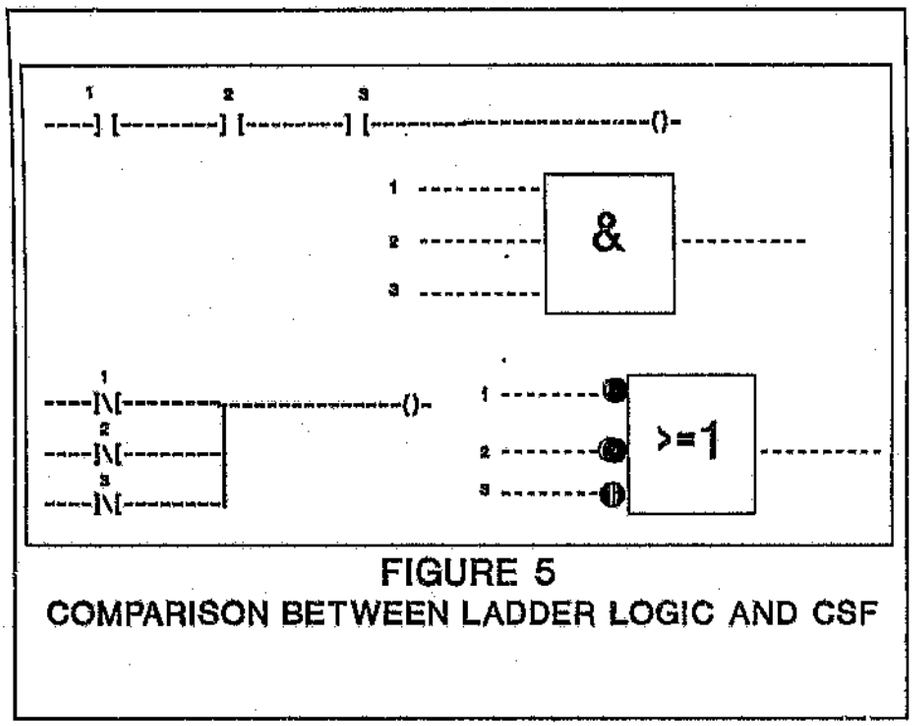


FIGURE 5
COMPARISON BETWEEN LADDER LOGIC AND CSF

2.2.3 Boolean Logic

Boolean logic (BL) is a specialized assembler language that uses much of the syntax of an assembler language. Figure 6 shows the logic for the simple pump given in BL. Languages of this sort use much the same ideas as other graphical methods such as ladder or CSF. The statements follow along the lines of CSF in a written fashion. The AND function is as follows:

A (input 1
Input 2)
= Output 1

The above code 'ANDs' inputs 1 and 2 and places the result in output 1. Included in the statements are the following AN (NAND),O (OR), ON (NOR) etc.

Some PLCs do not allow for the handling of analog I/O within the framework of the graphical language and therefore analog handling and calculations must be done using assembler type statements. BL allows for conditional execution of certain conditions and calling of subroutines.

```
0000    : A (
0001    : A FIELD START
0002    : AN RUNNING
0003    : O
0004    : A START
0005    : AN TEMP FLAG
0006    : )
0007    : A TRIPS
0008    : A STOPS
0009    : = START
000A    : ****

000B    : A START
000C    : L 5 SECS
000E    : START TIMER 1
OOOF    : A RUNNING
0011    : RESET TIMER 1
0012    : A TIMER 1
0013    : = TEMP FLAG
```

FIGURE 6
BOOLEAN LOGIC OF A SIMPLE PUMP

One of the major disadvantages of BL is that as it is not graphical it requires more thought when debugging as the function of each group of statements in not immediately evident. This is one area where the more classical area of program documentation is very much a necessity and not a luxury.

Programming in BL also requires more translation between design and implementation and a knowledgeable maintenance staff. BL is fairly simple for an experienced assembler programmer to use and is closer to classical programming techniques than any graphical language.

2.2.4 Sequential Function Chart

Sequential function chart (SFC) is a graphical method of functional analysis. SFC is marketed under various names such as Grafcet (Telemechanique 1987) or Graph 5(Siemens 1986).

SFC represents functions of a sequential automated system as a sequence of steps or transitions. Graphical programming was specifically developed for sequential control problems where the steps are sequential or time dependent. (Tinham 1986) Each step may represent a command or action that is either a logical 1 or 0. The flow of control is governed by a conditional transition that is either true or false. When a transition condition becomes true control is passed from one step to the next. SFC is a higher level language than any of those discussed previously and is used as an outer shell from which lower level subroutines are called. Figure 7 shows the basic SFC symbols and gives a description of their meaning.

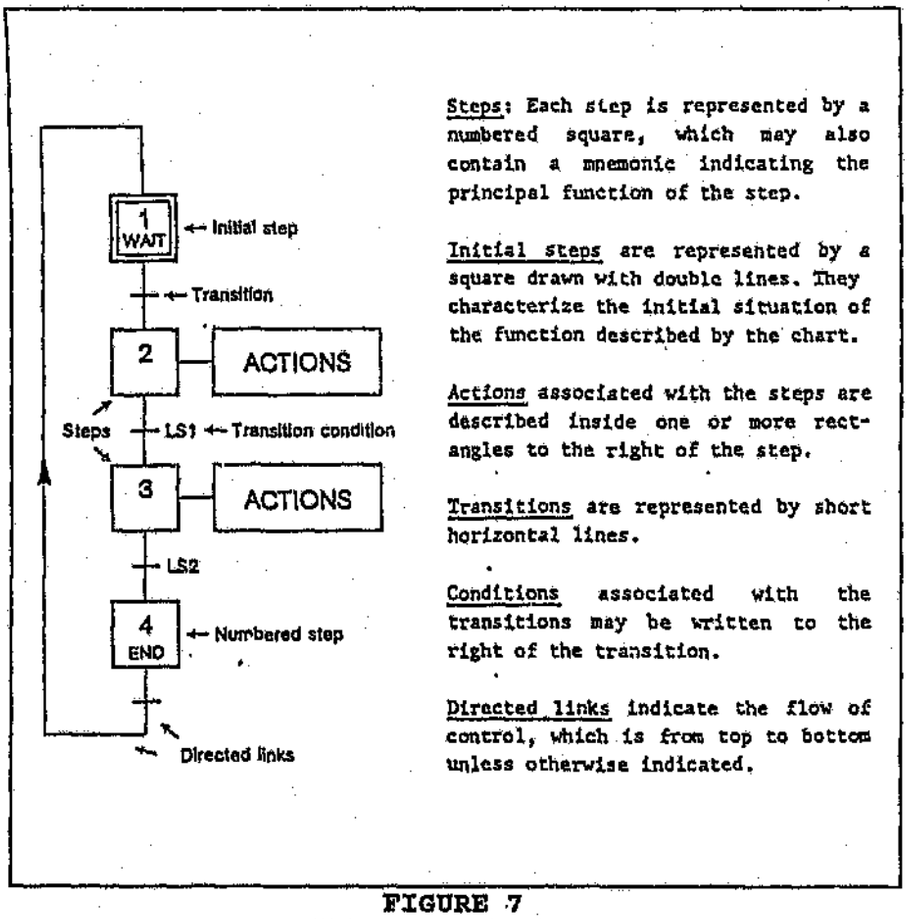


FIGURE 7
BASIC SFC SYMBOLS
(SOURCE : TELEMECHANIQUE 1987)

The best method of describing SFC is by example. This is given in figure 8 (Telemechanique 1987) In this example a circular plate is rotated through 120 degrees in order to serve three workstations at which the following operations are performed:

Station 1 - Loading the workpiece

Station 2 - Clamping and drilling

Station 3 - Testing by depth gauge and removal

At station 3 the depth gauge must descend to its lowest position otherwise the machine will stop. This will require the operator to remove the defective piece and restart the system. Figure 9 shows the SFC chart for this operation. This shows that after the initial operation the three steps are simultaneously activated each continues independently until steps 4,9 and 14. Here the program waits until all three sequences have been completed until continuing to rotate the plate. From this it can clearly be seen that a block represents an action or group of actions. Transition from one block to another is based generally on satisfying some condition.

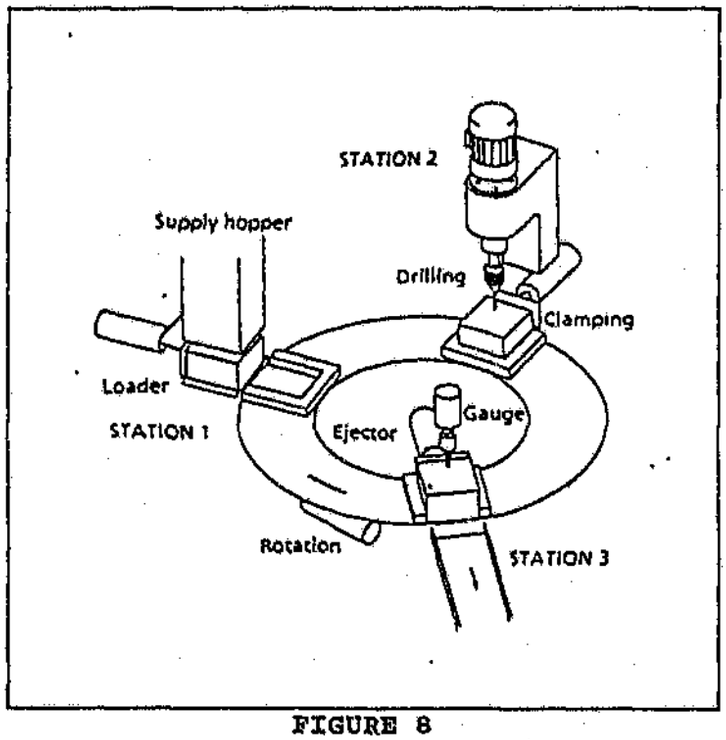


FIGURE 8
DRILLING STATION LAYOUT
(SOURCE : TELEMECHANIQUE 1987)

SFC follows closely along the lines of classical structured programming as each of the blocks shown could be an entire subroutine , function or procedure. SFC

also serves the purpose of hiding of the actual bit manipulation of each device. This becomes a good feature when dealing with large programs as a block on the outermost SFC chart could represent another SFC chart which in turn could be made up of SFC charts, procedures or a simple statement in ladder, STL or CSF. An example of this is shown in Figure 10.

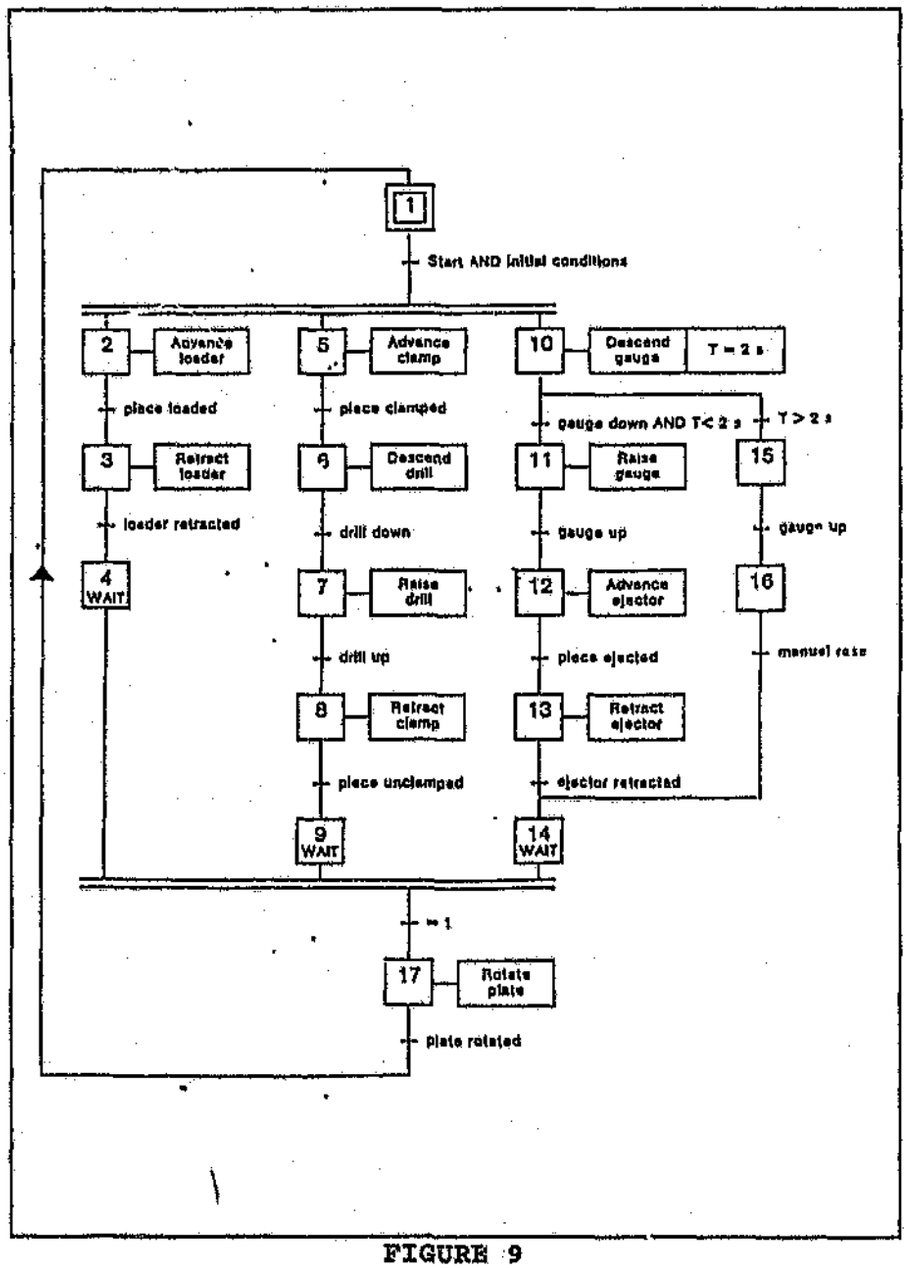


FIGURE 9
SFC CHART FOR DRILLING STATION EXAMPLE
(SOURCE : TELEMECHANIQUE 1987)

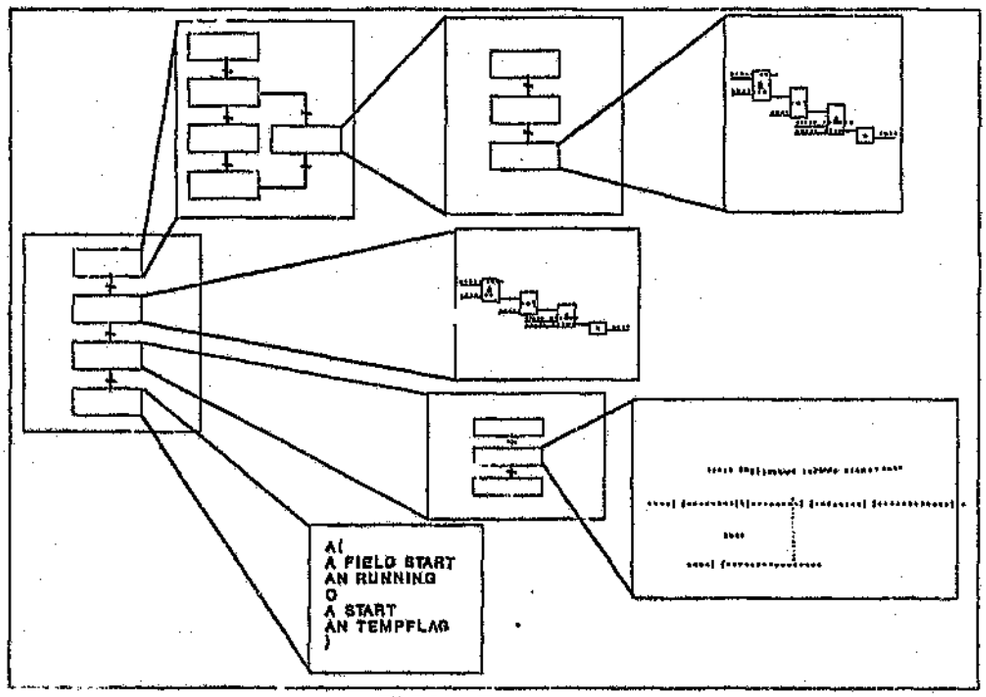


FIGURE 10
SFC CHART MADE UP OF OTHER LANGUAGES

2.3 Conclusion

PLC languages have progressed a long way from the early conversions from relay panels. This is however just the tip of the iceberg as the predictable trend in the future will be to high level languages such as FORTH and C. Some vendors are already offering limited versions of these languages for the programming. This type of development will allow the power of design methods such as Object Oriented design to be used to their fullest extent. This has however the disadvantage that users of programs will require far more training in the different languages. In addition this places a burden on designers and programmers for more comprehensive documentation and program commenting. These minor disadvantages will definitely be outweighed by the benefits obtained in terms of overall program complexity and programmer utilization. It is hoped that these advancements in the PLC field will lead to their wider use in even more complex control environments.

An area that was discussed in chapter 1 is that of DCS systems. The classical PLC programming approaches have to a large extent fallen away with the advent of the DCS. This is evidenced in the shift from "programming" to "configuration". However, the design methodologies are still relevant to any type of control environment, only the programming interface has changed.

3.0 OBJECT ORIENTED SOFTWARE DEVELOPMENT TECHNIQUES

"Object oriented development is an approach to software design in which the decomposition of a system is based on the concept of an object. An object is an entity whose behavior is characterized by the actions it suffers and requires of other objects."
G BOOCH 1986

There are as many views on what Object-oriented programming is as there are computer scientists (Pascoe 1986). This is largely due to the origins of the concept of Object-oriented programming techniques. The term was originally used to describe the Smalltalk programming environment developed at Xerox PARC. (Goldberg et al 1983) Today it describes a completely new dimension in programming and a vastly improved method of developing software and increasing programmer productivity. (Bright 1988)

The simplest definition of object oriented languages is that they allow the programmer the ability to define a new class. (Bright 1988) A class is a new type of data. The compiler then treats the new class as if it was part of the existing data type definition of the language. To be classified an object oriented language, an language must exhibit four characteristics viz. data abstraction, dynamic binding, information hiding and inheritance. (Pascoe 1986) A number of high level languages have been mistakenly called object oriented as they do not exhibit all four characteristics.

3.1 Data Abstraction

The process of defining new data types or classes is referred to as data abstraction (Pascoe 1986) or abstract data typing (Bright 1988). A programmer defines the class along with the internal method of representing that class. This internal representation may consist of basic data types (integer, string real etc.) or other class types. In addition to the definition of this class, the programmer defines a set of manipulative procedures to access and manipulate that data. An example of data abstraction considers the definition of two data types viz Stack and list. Each type has differences in their methods of data storage and manipulation. Therefore each will have a set of procedures that will manipulate the associated data type i.e. Initialize or print. This has as the disadvantage that the overall structure will be larger due to calling separate manipulative procedures. However, this does allow for ease of definition of new classes. Assume that the types List and Stack are defined, the type Queue could now be defined along with its set of internal manipulative procedures which would be independent of the existing data types.

This aspect of object oriented design forms the basis of using these techniques for the development of control system software. The fundamental reason for this is that the classical control problem revolves around the controlling of a few fundamental types of equipment, an ideal situation for the classing of types.

3.2 Information Hiding

Information hiding considers the aspect that the state of a software module is only visible from within that module. Thus data is manipulated only by internal procedures. This feature allows for the efficient design of modules such that the internal variables within a module are not accessible from outside the module. In effect internal data structures may be changed without affecting other modules in that implementation. Most programming languages support information hiding except Pascal as it does not allow a method of defining static data types within a procedure.

Information hiding could play a vital role in the safety aspect of control systems. All of the information regarding a piece of equipment within a plant can be hidden, and only accessible to the piece of control code that deals with that equipment. This would ensure that each piece of code only deals with the sections of memory and variables associated with a particular object, and does not have global access. This aspect will be discussed in chapter 5.

3.3 Dynamic Binding.

The issue here is that the message initialize, for example, sent to different objects of each of a number of different classes will elicit different responses from each of them. Thus the initialization procedure for the stack and the list will be substantially different, but the message to tell the variables what to do is exactly the same.

This area is often referred to as polymorphism (Booch 1986), dynamic binding (Pascoe 1986) or commonality (Bright 1988). Polymorphism is the ability of objects of different classes to behave in the same way, whereas dynamic binding is the mechanism for selection of the correct behavior at runtime. The advantage of this occurs where a new data manipulation procedure needs to be added to a program. Assume the procedure **"list_to_Screen"** is to be added. In this case there may be numerous data types i.e. integers, strings and arrays. Traditionally the new procedure would have to cater for all the existing data types and elicit the corresponding responses. It is potentially dangerous to attempt to list an array in a procedure designed to list integers. In addition if a new data type such as two dimensional array is added to the program then all the existing procedures would have to be modified to accommodate this data type, including the list procedure.

Object oriented programming pushes the responsibility of type determination onto the data types themselves. Each type of object will be sent exactly the same message and each will then display the correct behavior for that type of object.

Simply defined the area of dynamic binding allocates responsibility. **"An object decides, at runtime, what to do with the message you've sent it."** (Bright 1988)

3.4 Inheritance

Inheritance enables programmers to create classes and sub classes. This is called subclassing where the lower level new object is called a subclass of the superclass. The subclass inherits its behavior from the superclass and will add aspects of its own to the characteristics of the superclass. Inheritance allows programmers to create new classes of objects by specifying the differences between them and the original class, instead of starting from scratch. This is associated with another area, that of software reusability, where sections of code an be reused many times. (Meyer 1987). Objects are defined as being either a subclass or a variable, which is defined as being of a class type.

Bright defines inheritance or **"type derivation"** as **"define a new class which is this old class here plus some additions and changes I want to make"**

Once again, this becomes relevant in the control system environment, where all types of electrical equipment may behave in the same fundamental way. However, each may exhibit slightly different characteristics. Thus some of the classes may be derived form a base class. This forms the fundamental basis of the derivations in chapter 5.

3.5 Design methodology.

One aspect that requires consideration is the methodology of linking of objects into a piece of coherent code. Jamsa (1984) considers that the emphasis of object oriented design is to **"keep the solution as near to the real world problem as possible"**. The methodology proposes that the clear statement of the problem can be utilized to produce the implementable units in the identification phase of development. When considering the problem statement it will become evident that the objects are the nouns and that the operations performed on them are the verbs.

Jamsa proposes the following steps in the design procedure :

1) Define the problem
2) Develop an informal strategy
3) Formalize the strategy
    a) Identify the objects and their attributes.
    b) Identify the operations on the objects.
    c) Establish the visibility of objects.
    d) Establish the interfaces between objects.
    e) Implement the operations of each object.

The first step is a clear concise plain language definition of the problem. This should include a full description of all the objects and a definition of the basis for their interaction. In an industrial application this would most probably take the

form of a functional description of the system. This functional description or specification will in the majority of cases include a description of the interlocking between devices and an indication as to the controllability of the system under varying conditions.

The informal strategy phase considers the development of a process description. This, unlike the problem definition is not a description of the process but rather a software description. Jamsa (1984) suggests that structured English could be employed at this stage. A useful tool at this point in the development life cycle is perhaps Program description language (PDL). This is discussed in detail by Walker (1984) and Caine (1980, This PDL description can be used as a definite starting point for the des.. The one problem with PDL in this context is that it follows very closely along the lines of Pascal, a language that is not really suited to the object oriented environment. However, a designer may make use of another suitable type of pseudo code, perhaps along the lines of C++.

The formalization of the st y follows along similar lines to those proposed by Abbott (1980) and discussed by Booch (1986). The first step i.e. identification of the objects and their attributes, involves the recognition of the major actors, servers and agents within the framework of the problem. One method that has been proposed is to consider the problem definition and extract all the nouns from the description. These will be the major objects in the problem space. At this stage it may also be found that there are objects that share many attributes. These can then be combined to form a class of objects and a corresponding set

of subclasses. This will allow each object to be considered as an instance of a class.

Identification of the operations suffered by and required of objects forms the next step in the formalization of the strategy. This serves as a method of characterizing the behavior of the objects and classes of objects. Here the "**static semantics**" (Booch 1986) of the objects are determined by establishing which operations may meaningfully be performed on that object. Considering the method discussed above if all the verbs are extracted from the description then these will define the actions suffered by objects. These action by definition then define the actions of an object when acted upon by other objects.

The establishment of the visibility of objects establishes the static dependence of objects upon each other. This step serves to determine which objects are seen by other objects. The fourth step, the establishment of interfaces, involves the production of a module or object specification. This requires combining all the data captured in the previous steps and assembling it into a coherent specification. This information serves as the basis for establishing the boundary between internal and external views of an object.

The fifth and final step involves the implementation of each object using available language resources. At this point a suitable representation for classes and subclasses of objects is found. If an object is found to be composed of several subordinate classes of objects then the entire method is repeated to further

decompose the object. However in the majority of cases an object is built up by using one of the existing classes and subclassing, thereby adding characteristics of its own the existing superclass.

3.6 Conclusion

This chapter considered the mechanics of Object-oriented design of software. Object-oriented development will undoubtedly have major implications in the future of software engineering due to the large number of features this type of methodology encompasses. These steps discussed throughout the chapter constitute the lifecycle of the software development. Booch (1986) states that " **Object oriented development is a partial lifecycle method: It focuses on the design and implementation stages of software development**". The steps listed above may appear mechanical, however Abbott observes that it is not an automatic procedure and requires **"a great deal of real world knowledge and intuitive understanding of the problem"**. However, this greater understanding will help to simplify software and allow for greater programmer productivity and utilization of resources. It is believed that the principles of object-oriented design will form the fundamental basis for a large number of software development applications in the future. The next chapter considers the application of Object-oriented design techniques to the development of generalized PLC software. This will show the versatility of the design methods and how their usage can be modified to encompass other areas of software development.

4.0 CONTROL SYSTEM SOFTWARE DESIGN

4.1 Introduction

This chapter covers the design of control system software using the principles of object oriented design. It is at this point that a distinction should be made between Object oriented programming and Object oriented software design. Duff (1986) Cox (1986) and Pascoe (1986) are amongst many authors that are concerned with language specific issues. They consider the pros and cons of each language and its possible use for implementing software designed using object oriented design techniques. The emphasis lies with the inherent features of the languages, such as Smalltalk, that will allow the characteristics discussed in chapter three to be implemented to their fullest extent.

On the other hand Booch (1986) places his emphasis on the necessary development principles and briefly discusses the implementation of his ideas in a specific language environment. Bright (1988) discusses implementation issues in a language specific environment. This is the area that will be the focus of this chapter i.e. that the design techniques associated with object oriented software development are sound and can be utilized in a language independent fashion. This chapter will consider the application of the techniques discussed in the previous chapter with regard to their usage in the design of industrial control systems.

At the outset it should be noted that PLC languages suffer from many constraints when applied to pure object oriented development. However the nature of the dynamics of the system to be controlled will allow for the implementation of these Object oriented techniques. Certain aspects of the design phase will become fairly abstract but will nevertheless allow for the more efficient design of industrial control system software

Current trends in PLC languages have been previously considered. A few PLC vendors are already heading away from the classical PLC programming languages and are allowing for software to be written in languages such as C and Forth. There are C++ compilers that have an intermediate stage of compilation that generates a set of standard ANSI C procedures. These could then be downloaded to a PLC that is capable of being programmed in C. This will ease the burden of implementation and the level of programming will be much better. These languages are more suited to implementation of Object oriented software. As they gain wider acceptance, the principles of object oriented software design will become necessary design tools.

4.2 Generalized Software Design

Consider the design of a generalized control system. There will generally be an overall control philosophy that will define the basic rules of operation of the major pieces of equipment and equipment types. This is often referred to as the MCC (Motor Control Center) philosophy. This philosophy does not cover each individual piece of equipment, but rather the control aspects of types of equipment and the interactions between these. The control system philosophy will define the necessary classes and subclasses to for the programmer. The functional specification of the control system will define the specific requirements of each piece of equipment along with the physical data. The latter aspect covers the information such as what connection from the PLC will go to that specific piece of equipment.

For the purposes of example, all equipment that is classified as electric drives will be considered. In an industrial application this will include pumps, motors, conveyor belts etc. These may be of different types and sizes but the base assumption is that they all operate in similar fashions.

A simple operating philosophy was discussed previously, but can be summarized as follows : Upon receiving a request to start signal the control system checks whether the drive's trips and stops are in a healthy state and whether or not it is running. If these conditions are true then in the most basic case a signal is sent to the drive to allow it to start.

According to the principles of object oriented design a superclass, perhaps of the type **"drive"** could be created. This would be the simplest form of logic. Figure 11 shows the logic of this simple **"drive"**.

As each different drive is encountered, which differs slightly from the superclass, a new sub class can be created to cater for that type of drive and its associated extra logic.

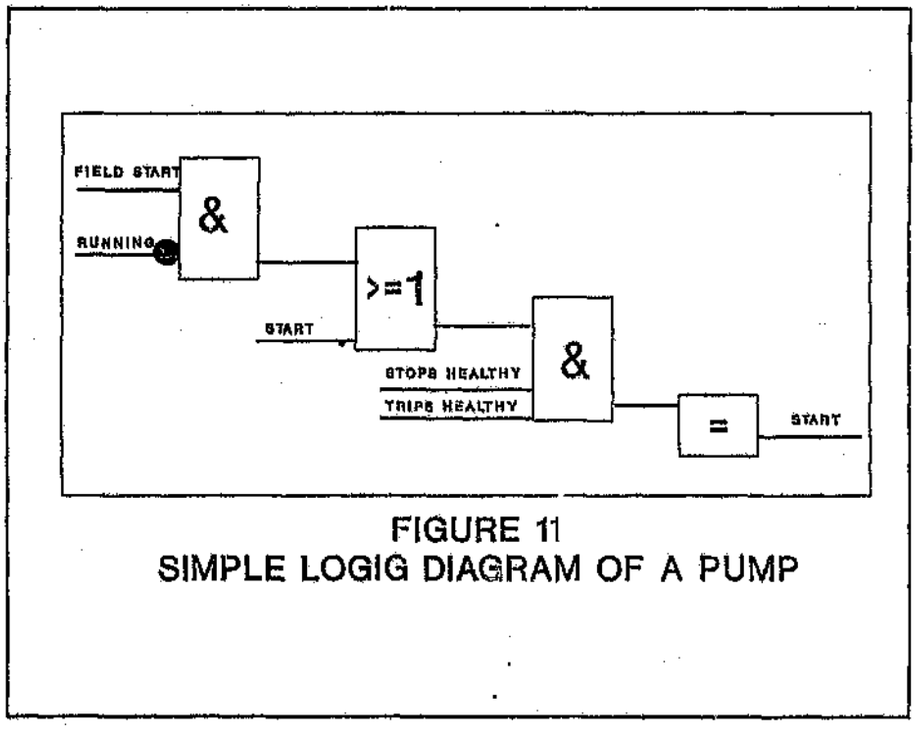


FIGURE 11
SIMPLE LOGIC DIAGRAM OF A PUMP

Assume for the purposes of example that there exists a drive that when a request to start signal is received will sound a siren for a certain time period before starting. This siren is called a safe start siren and is used to warn

personnel in the area of the impending start. In this case a subclass of **"drive with_siren"** could be created to cater for this instance of drive.

The logic diagram of this **"drive_with_siren"** is shown in Figure 12. A timer is used, that times the siren. Upon completion of the timing period the timer output will be turned on, and paralleled with the rest of the logic.

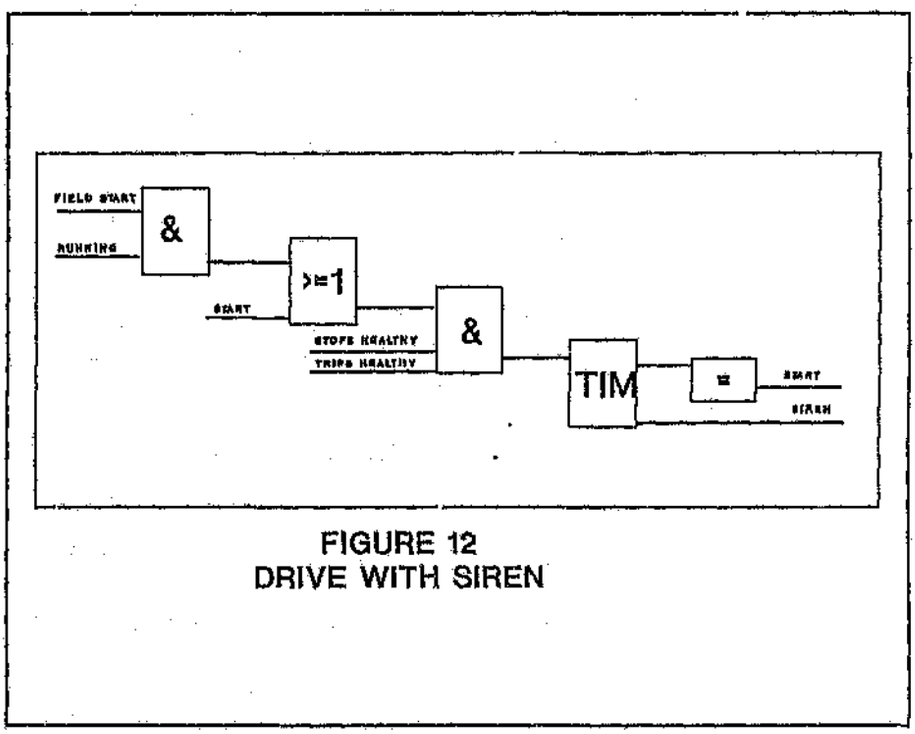


FIGURE 12
DRIVE WITH SIREN

In most applications there will also be numerous interlocks between the drives. These will prevent certain areas of the system from starting before it is safe to do so, or in the event of a device tripping then the associated devices are required to stop. The methodology of coping with this is to have an internal variable within a class **"interlock"**. This class of interlock can then be created which will cater for an open-ended number of interlocks for each drive. Each

drive routine will have a specific interlock object passing, as it were, a clearance signal to the control module to allow it to continue and start the drive. The logic diagram illustrating this logic is shown in Figure 13.

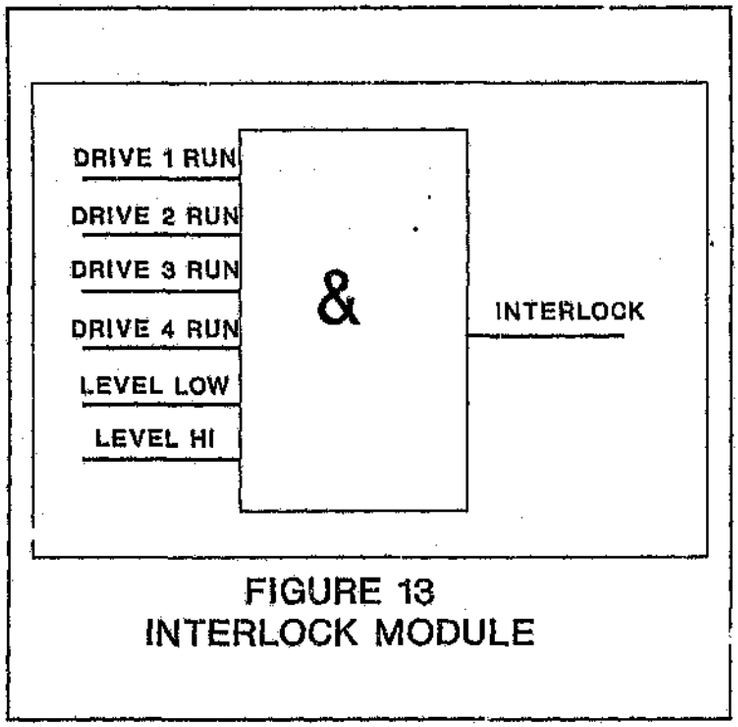


FIGURE 13
INTERLOCK MODULE

Each class and subclass of drive will respond in a different way to the same message. The superclass of drive will merely start the drive. The **"drive_with siren "** will sound the siren before starting the drive. The **"Drive with interlocks"** will first get a clearance before starting the drive. This action of eliciting differing responses to the same command fulfills the object oriented characteristic of dynamic binding. By virtue of the definition attached to the superclass of **"drive"**, the subclasses obey the requirements of inheritance.

Data abstraction is considered in the aspect of the design, that a few standard classes can be designed, and the rest of the classes formed form subclasses of

these. Information hiding is used to protect the traditionally global variables from being modified by the program, in areas where it should not be affected.on hiding become the most complex to deal with when in the area PLC software. This serves merely to emphasize once again that the objective of this dissertation is to consider the usefulness of object oriented design techniques as a method for the development of control system software. It is conceivable that in the not too distant future there will be PLCs using languages that can fully support and make use of the features of Object oriented design techniques.

4.3 Conclusion

The principles of Object-oriented design are sound when applied to the development of software in an high level environment. However when applied to PLC software the techniques have to be addressed in a slightly different fashion. The objectives of this dissertation are to consider the use of object oriented techniques within the framework of the PLC software development environment. This chapter has considered how the four major areas of definition of Object-oriented design can relate to PLC programming. There are a number of obvious constraints when considering the PLC software and it is accepted that PLC languages will not currently fully support these high level design techniques. However in the foreseeable future there will be PLC programming languages that support the features of Object oriented design. It is at this time that Object-oriented design methods will be used to their fullest extent. In the interim period the applicable features can be utilized to enhance the software development lifecycle.

5.0 SIMPLE OBJECT ORIENTED IMPLEMENTATION

5.1 Introduction

The previous chapter considered the implementation of the simple operations associated with electric drives. It also discussed the possibilities of using object oriented approaches in the development of the software to control this type of equipment. This chapter continues along this track, with the implementation of the software in C++. This chapter also considers the interface to the physical world and how to overcome the limitations therein.

This chapter will also begin demonstrating the power of object oriented design and how all four characteristics thereof are used within a practical example. These four aspects were discussed in chapter 3, and are be used as guiding principles throughout all the software design.

5.2 Physical Input and Output

One of the major areas of consideration must be the comparison of the object oriented software to the physical PLC. All PLC systems work on the principle of having a series of Input/Output (I/O) racks. Each of these makes a physical connection the pieces of equipment that are to be controlled. In chapter 2, when the input signals (field start, trips healthy etc.) were discussed, these referred to physical signals coming into one of the I/O cards within an I/O rack. The software written for this dissertation will not actually operate on a PLC as no PLC can currently operate with an object oriented language. Therefore the problem lies in defining I/O and defining a method of mapping such I/O into the real world.

The solution to this is the creation of a number of classes called IO. These represents the physical I/O points which receive and send signals to and from the plant.

There are four types of physical I/O associated with a PLC. These are :

Digital inputs (DigI)

This would be a binary signal coming into the PLC.

Digital outputs (DigO)

Similar to the inputs but being sent from the PLC to the equipment being controlled.

Analog inputs (AnI)

This is an analog signal (such as a tank level) being sent to the PLC from the actual unit.

Analog outputs (AnO)

These are analog signals sent by the PLC to specific units. These could be to control the opening of a valve or the speed of a variable speed motor.

The features of object oriented languages allow for the definition of these classes to prevent the PLC program attempting to assign a value to an input or visa versa.

Thus at the most fundamental level there would be two classes defined. These would be DigO and DigI.

The definitions (in C++) would be :

```
class DigO
{
  public:
     int state;  // digital state
     char* address; // address of PLC IO point
     DigO(); // constructor function
};

class DigI
{
  int state;
  public:

     char* address;
     DigI();
     void setDI(int value); // function to set value of private variable
     int readDI(); // function to read value of private variable
};
```

This definition requires some explanation, as there are some areas where the principle of information hiding is used. This is done in an attempt to map the somewhat contrived situation into the real world. The digital output signals associated with a drive are public domain in most PLC's. Thus they can be modified by any procedure operating anywhere within the scope of the PLC. This issue will be addressed later when the definition of the IO rack is given. However, it suffices to say, that the class DigO (digital out) is made up of two member variables and a constructor function. The two member variables are the state of that output signal and the address of that digital output. The latter refers to the address that is used by the PLC programmer to assign a physical IO point to a memory location. Generally, the physical IO is mapped into a segment of the PLC memory. This facilitates the quick access of the value, rather that having to read from the physical IO card all the time. The constructor function is used to initialize the state of the objects of this class, when an object is defined.

The definition of the digital input (DigI) is rather more complex. A large number of PLC's do not allow for easy access to digital input values. This implies that the program as it stands cannot overwrite the input area of memory. This is obviously to safeguard against the problem of the PLC writing the wrong value in place of a physical input. To cater for this feature, the state of the DigI has been defined as private. This means that only member functions of the class DigI can access and modify that value. It must be stressed that in this application, there is a tradeoff between 'good' object oriented principles, and mapping into the real world. In the majority of PLC systems there are very few 'private'

variables. Thus, the use of the object oriented principle of information hiding is used to a limited extent. the If this program was being written for an actual PLC then there would be no need to have all of the member functions that have been written. There is a void function called SetDI which is used to force the input to a specific value. This is used as there is no actual IO point available. The integer function ReadDI is used to ascertain the value of the private variable state. This function would also not normally exist as the access to the input values would normally be unrestricted in a reading mode. Thus right at the outset the program is beginning to use features of object oriented design.

The full code of the member functions is given below :

```
// Constructor functions :

DigO::DigO()
{
     state = 0;
}

DigI::DigI()
{
   state = 0;
}

// Functions for using the state of Dig I

void DigI::setDI( int value)
{
  state = value;
}

DigI::readDI()
{
return(state);
}
```

The code segments given above are part of the file "digital.hpp".

As previously stated, the IO signals to the PLC are grouped into IO cards and racks. Thus the next class definition is that of an IOrack.

This is merely a **"holder"** of IOs. The IOrack class serves as a type of interface to the real world. It contains the most basic IO required for the operation of a general electric drive.

```
class IOrack
{
   public:
      DigO start, localstart;
      Digl seqstart,fieldstart,tripshealthy, stopshealthy, running;
      IOrack();
};
```

The class defined by IOrack does not contain any logic. It merely groups the IO together and is used to relate specific IO points to a specific drive. When a variable of class IOrack is defined, the specified IO are defined. The definition section of the class definition calls the correct IO types and initializes the IO accordingly. The constructor function IOrack() required at start-up to facilitate the definition of a new variable of the class IO RAck. The code segment of this function is empty, as when an IO rack is called it creates all the variables which initialize themselves.

The variable localstart is used as a temporary carrier of information between the super- and subclasses. The base type of IO rack contains the most basic of IO and does not cater for any new types of drive IO. These would be made local variables within the new class itself.

5.2 General Drive Class

Since the simplest classes necessary have been defined, the basic electric drive operations can now be defined. These are contained in the class General Drive.

The structure of the code is such that the class of General drive forms the heart of the all the drive related operations. All the basic IO, procedures and operations are defined within this class. All actual operating drive classes are derived form this base class.

Certain functions are defined as virtual, and this allows for function overloading. This is perhaps one of the most fundamental benefits of using Object Oriented design and associated languages. The same function may be used to denote a number of different operations. The language will determine at run time which of the functions to use. This allows each of the sub classes to behave in fundamentally different ways. The listing of the definition of the class is given below. Following this is a brief description of each of the lines in the class definition.

```
class GeneralDrive
{
  public:
      char namestr[20]; // drive name
      char typename[30]; // drive type
      GeneralDrive * nextptr; // pointer for next drive in list
      Interlock inter-lock; // interlocks of this drive
      addInterlocks(); // manipulating interlocks
      delInterlocks(); // as above
      int type; // used to globally define drive type
      virtual status(); // prints out all drive IO statusses
      virtual addtolist(char name[20]); // list manipulation
      setrunning(); // simulate PLC action
```

```
    IOrack rack; // IO of drive
    virtual operate( int value, int error =0); // specific operations
    operate1(int value, int error =0);  /* basic operations */
    GeneralDrive(); // constructor
};
```

The two character strings namestr and typename are used in defining the uniqueness of the drive. The type name is applied principally in the definition of new drive types, where a list of drives if formed and the types are listed for the user. This will be explained in detail later.

The pointer to the class called nextptr is the pointer that forms the heart of a linked list of drives. The only way of allowing for the creation of drives at run time was to use the linked list approach. Essentially, during run time all the existing drives form part of a linked list of drives anchored by the pointer first. This shows another fundamental principle of object oriented design, in that the pointer is to the superclass type, but the linked list can be heterogeneous and made of any of the subclass types. This allows for continuity in the program and reduces the overhead in the program of maintaining a number of lists for each subclass.

The interlock will be discussed in detail, when its own class is discussed. However, for completeness, the interlock relates to the safety features in the plant and unless the interlock conditior are met the drive will not be allowed to start.

The member functions add- and del-interlocks are used within the program to set and delete interlocks associated with a drive.

The integer type, is used to distinguish the different classes from each other. This is primarily used in creating drives at run time. The function is to eliminate the hard wired code generally associated with the selection of different types from a list of choices. Once again this will be described in more detail when functions that use this aspect are discussed.

The member function status is required to generate output to the screen of the status of all the input and output variables associated with a drive. This is used in a number of instances where a display of this nature is required. This is also defined as a virtual function, and therefore it allows a new subclass to redefine its member function called status. This is required if at a later date a sub class is defined that has fundamentally different output requirements. If the redefinition is not present, then the base class function will be called.

Addtolist is once again a function that is used in fundamentally different ways by each subclass. Its prime function is to add a new drive of a specific type to the linked list of drives.

Setrunning is a member function common to all subclasses and is used to actually determine if the drive is running.

Rack is a variable of type IORack. It contains the basic IO defined for a particular class. All IO operations are carried out using this IOrack. This is a very sound form of protection in PLC programming. It is often that the programmer types in

the wrong mnemonic for a variable and thus struggles to find an error. The fact that the IO associated with a drive is only accessible to that drive makes it impossible to access the IO associated with another drive. The additional benefit is that when programming all the names of the variables are the same, unlike conventional programs where variables are addressed by their position on the physical IO card.

The two member functions operate and operate1 form the heart of the drive operation. Operate is a common function that defines all the necessary basic drive operations. Operate1 is specific to each drive type and is called from operate. The same effect could have been achieved only one virtual function, and defining absolutely unique operations for each drive type. It was decided to use a common operating core and allow each drive to configure its own specific parameters Thus when a drive is passed the message operate it calls operate1 which then carries out the core drive operation. This will be expounded upon later in the descriptions of the various drive types. The IO point local start is used as a transfer mechanism between the procedures.

The last member function of the class General Drive is the constructor function. This is called each time a drive is created and is used for internal memory allocation.

The following few pages list the actual code for each of the member functions of the class General Drive.

```
/* Initialization calls the other defined classes and allows them
   to initialize themselves */

GeneralDrive::GeneralDrive()
{
        nextptr = NULL; // perhaps superflous , but for initialization
}

/* Operate1 forms the heart of the operation. It specifies the most
   basic drive operations  */

GeneralDrive::operate(int value, int error)
{  }

GeneralDrive::addtolist(char name[20])
{   }

GeneralDrive::operate1(int value, int error)
{  timet t0, t1;
   int temp;
   switch (value)  {
      case(0):break;
      case(1):if (rack.fieldstart.readDI()==0) // check field start buttin
                 { temp=1;}
               else {temp = 0;}
               rack.fieldstart.setDI(temp);
               break;
      case(2):if (rack.tripshealthy.readDI()==0){ // check trips and change state
                 temp=1;   }
                 else {temp=0;}
                 rack.tripshealthy.setDI(temp);
                 break;
      case(3):if (rack.stopshealthy.readDI()==0){ // check stops and change state
                 temp=1;}
                 else {temp= 0;}
                 rack.stopshealthy.setDI(temp);
                 break;
      default:error =1;
                  }

\\ next code is actual drive logic

  if (rack.seqstart.readDI()==1) {rack.fieldstart.setDI(1);}
        r a c k . l o c a l s t a r t . s t a t e        =
((((rack.fieldstart.readDI()==1)&&(rack.running.readDI()==0))||(rack.localstart.state==1))&&
                          (interlock.setstatus()==1)&&
                         ((rack.tripshealthy.readDI()==1)&&(rack.stopshealthy.readDI()==1)));
  if (rack.seqstart.readDI()==1) {rack.fieldstart.setDI(0);}
}

\\ used to set state of running signal of drive
GeneralDrive::setrunning()
{
   int temp;
    temp  =((rack.tripshealthy.readDI())&&(rack.stopshealthy.readDI())&&(rack.start.state));
    rack.running.setDI(temp);
}
```

```
\\ prints out status of drive IO
GeneralDrive::status()

{
        printf("%s\n",namestr);
        printf(" field start %d",rack.fieldstart.readDI());
        printf(" trips        %d",rack.tripshealthy.readDI());
        printf(" stops        %d\n",rack.stopshealthy.readDI());
        printf(" seq start    %d",rack.seqstart.readDI());
        printf(" running      %d",rack.running.readDI());
        printf(" start out    %d\n",rack.start.state);
}

\\ used to add drives to the interlock list
GeneralDrive::addinterlocks()
{
  GeneralDrive * tmpdrive;
  int select,no;
  interlocknode * currentnode;
  interlocknode * temp;
        do
         {
          current = first;
          no = listdrives();
          printf("\n Which drive do you want to add to the interlock ( 0 to quit): ");
          scanf("%d",&select);
            if ((select > 0) & (select < no))
                {
                  currentnode = interl[illegible]de;
                   while ( currentno[illegible]
                        {
                          temp = cu[illegible]ode;
                          currentnod[illegible] = [illegible]urrentnode->next;
                        }
                     currentnode = temp;
                     current = first;
                     current = finddrives(select);
                  interlocknode * nextnode = new interlocknode;
                  currentnode->next = ne[illegible]node;
                  (nextnode->runstate)
                          =&(current->nextptr->rack.start.state);
              strcpy(nextnode->name,current->nextptr->namestr);
                  nextnode->next=NULL;
                  currentnode = nextnode;
                  }
         } while (select > 0);
}
```

```
\\ used to delete drives from the interlock list
GeneralDrive::delInterlocks()
{
  int select,no;
  interlocknode * currentnode;
  interlocknode * temp;
  interlocknode * temp2;

        do
         {
          current = first;
          interlock.list(&no);
          printf("\n Which drive do you want to delete from the interlock ( 0 to quit): ");
          scanf("%d",&select);
            if ((select > 0) & (select < no))
                {
                  current = first;
                  current = finddrives(select);
                  currentnode = interlock.firstnode;
                    while    ((    currentnode    !=
NULL)&(strcmp(current->nextptr->namestr,currentnode->name)!=0))
                        {
                          temp = currentnode;
                          currentnode = currentnode->next;
                        }
                          temp2 = temp;
                          temp->next = currentnode->next;
                }
         } while (select > 0);
}
```

The case statement at the beginning of the operate function would not appear in a PLC program. This is used to determine changes in input state and then act accordingly, based upon the state changes. In essence this forms the simulation of the physical IO. The last line, also forms part of this operation as it emulates the action of the actual drive, by checking whether it can start, and if so, starts running.

5.3 Additional Derived Classes

The basic classes have been defined and it is at this point that the principles of Object Oriented design come into their own. The previous chapter discussed various variants on the basic drive class. Object oriented design allows for these to be implemented with little additional code. The class of general drive was defined in such a manner that additional logic is placed in the subclass's operate function. The new class then calls operate1, and then adds any additional logic into its own operate function. This is demonstrated with a number of examples.

5.3.1 Drive with no features

There seems to be little rationale in defining a new subclass that behaves exactly as the superclass. However, the reason was to maintain a continuity in the method of defining new drive types. These are all defined by subclassing of the class general Drive. It was thus logical to subclass a drive of the most basic class in this fashion. This could also be compared to the situation where the most general drive class is compiled into a library and users wanting to write programs would define just their new class as a subclass of the predefined class. The definition of the drive class (called Drive none) is given below :

```
class Drivenone : public GeneralDrive
{
   public:
        operate( int value, int error =0);  /* calls operate1 */
        Drivenone() ;
        addtolist(char name[20]);
};
```

The code used in the implementation of the class is given below:

```
Drivenone::Drivenone()
{
     strncpy(typename,"Standard Drive",30);
     type = 1;
}
Drivenone::;operate(int value, int error)
{
  int temp;
    operate1(value,error);
    rack.start.state=rack.localstart.state;
    setrunning();
    if (value > 0)
     {
     status();
     }
}
Drivenone::addtolist(char name[20])
{
   temp = new Drivenone;
   strncpy(temp->namestr,name,20);
   temp->nextptr = NULL;
}
```

The two function operate and addtolist are defined in the base class as virtual functions and thus are redefined in this section of the code. All the other definitions that are discussed will follow a similar format. However each of these will contain a fuller description of the code segments.

5.3.2 Drive with start siren

The class of drive with start siren works on the principle that upon start-up of the drive, a siren is sounded to warn personnel of the impending start.

```
class Drivesiren  : public GeneralDrive
{
        sirentimer(int period);
public:
        DigO siren;
        operate(int c, int d=0);
        Drivesiren();
        addtolist(char name[20]);
};
```

This subclass has an additional IO point called siren. This would be used to actuate the physical siren in the plant. This is simulated using the C++ sound function in the code.

This simulation takes place in the private member function called siren timer. This function is only accessible to variables of this class type. This is an important safety feature as there is no chance of the siren being sounded accidentally as it can only be sounded by a drive of type siren.

The code for the member functions is given below:

```
Drivesiren::Drivesiren()
{
        strncpy(typename,"Drive with start siren ",30);
        type = 2;
}
Drivesiren::operate(int c, int b){
       operate1(c,0);
      if ((rack.localstart.state==1)&&(rack.running.readDI()==0))
      {  siren.state=1;
         sirentimer(3); // starts siren for a time period
         siren.state=0;

      }
         rack.start.state=rack.localstart.state;
         setrunning();
  if (c > 0)
     {
     status();
     }
}
```

```
Drivesiren::addtolist(char name[20])
{
   temp = new Drivesiren;
   strncpy(temp->namestr,name,20);
   temp->nextptr = NULL;
}

Drivesiren::sirentimer(int period)
{
  timet t0, t1;
  printf(" starting siren ");
  time (&t0);
  do
  {
    time (&t1);
    soundtone(5000,100,100);
  } while (t1 - t0 < period && t1 >= t0);
  printf(" siren finished ");
}
```

The addition of the logic is achieved by use of the variable local start. The function operate1 does not actually send out a signal to the real world, it merely sets a flag called local start. This is then used as the basis for the additional logic. The siren is now started before the start signal is set. This is also conditional on the local start signal having been set.

5.3.3 Drive with Safety Timer

Once again this type of drive is derived from the base general drive. The mode of operation here is to start a timer, if the conditions for starting are in order. If the drive starts within the timing period, then the timer is stopped. If it fails to start them the start signal is removed and the drive returns to the ready to start position. This is a safety precaution, to remove the start signal.

```
class Drivetimer  : public GeneralDrive {
public:
        operate(int c, int d=0);
        Drivetimer();
        addtolist(char name[20]);
        };
```

It can be seen that the definition of the sub classes is almost identical in all cases. They all have the same member functions and these contain all the differences in the operation.

The code for the drive with timer is given below :

```
Drivetimer::Drivetimer()
{
        strncpy(typename,"Drive with start timer",30);
        type = 3;
}

Drivetimer::operate(int c, int b){
      operate1(c,0);
      int stat,statuss,temp;
      timet t0, t1;
      if ((rack.localstart.state==1)&&(rack.running.readDI()==0))
   {

\\ the hit any key is used to simulate the drive action
        printf (" \nHit a key to start the drive or else timeout " );
        time (&t0);
         do
        {
          time (&t1);
          statuss = (kbhit()!=0);
          if (statuss)
          {
          temp = getch();
          }
          stat = statuss||(t1 - t0 > 3);
        } while (!(stat));
        if (t1 - t0 > 3)
        rack.localstart.state=0;
   }
          rack.start.state=rack.localstart.state;
          setrunning();
  if (c > 0)
     {
     status();
     }
}
Drivetimer::addtolist(char name[20])
{
   temp = new Drivetimer;
```

```
  strncpy(temp->namestr,name,20);
  temp->nextptr = NULL;
}
```

There is no actual plant to generate a return signal a .d confirm that the drive is running. Thus this is simulated by asking the user of the program the strike a key to emulate the action of the return signal. This would obviously not be used in the actual implementation in a PLC.

5.4 Conclusion

It can be seen from the above examples that the principles of object oriented design may be applied in many forms. The special features of the language can be successfully applied in the area of control system software. This chapter considered the heart of any control system, that of the control of the numerous drives in a plant. Only three types of drives have been defined, but it is evident that the process of defining additional drives is fairly trivial, given the infrastructure.

The objective of this dissertation is not to write PLC code, but rather to show how the basic principles of Object-oriented design can be applied to make programmers more productive. This is a major benefit as the human cost in most control system design and implementation far exceeds the hardware costs of that implementation.

The following chapter covers the implementation of the additional infrastructural code required to generate the simulation of a plant, its control system and a supervisory system within that plant.

Detailed listings of the code written, part of which was discussed in this chapter is given in Appendix 1.

6.0 ADDITIONAL INFRASTRUCTURAL SOFTWARE

6.1 Introduction

This chapters considers the additional code required to demonstrate the principles described in the previous chapters. One of the problems faced in attempting to demonstrate the major hypothesis of this dissertation is that there are no PLC systems that allow for programming in an Object-Oriented language. Thus an attempt to demonstrate the action of such a program must emulate a large portion of the action of the PLC. An additional problem is that there is no physical plant available to be controlled and as such the action of the plant must also be emulated to a degree, even at a crude level. The last area for consideration is that in most plants the operating staff do not have access to the control system and therefore all commands are carried out using some type of supervisory system. This action must also be emulated in order to demonstrate the principles of this dissertation.

It was for this reason that a large amount of infrastructural software had to be written to allow for the emulation of these external devices. An attempt was made to continue using the principles of Object-oriented design in writing this additional software.

This software can de divided into three main categories :

1) Maintenance Software used for the storage and retrieval of data. This includes the infrastructure for maintaining the linked lists used in the software.

2) Additional control system software used for the implementation of the sequencing of drives and the interlocking thereof.

3) Supervisory and plant software written to emulate the action of both of these.

Very little code has been written that does not fall into the category of Object-orientec, but in certain instances it was necessary to support one of the major functions.

This chapter will consider the code written, in its most general form. The full code listing is given in Appendix 1.

6.2 Data Management

The issue of data management is vital in any program. A large portion of the effort was put into the methods of managing the data that is generated by the program. It was decided to use the linked list approach to the data management. This relieved the need for hardwired code and allowed the facility for the creation of new variables of the predefined classes at run time as opposed to compile time. One of the principles of Object-oriented design is that there is very little hard wiring in the code. This implies that in this case, where the operator has a number of classes of object that can be created at run time there must be as little fixed code as possible. Thus the most fundamental list created within the code is a list consisting of one of each of the drive types. This is achieved by the function drivelist. This is called at the start of the program only and never again. Its function is to generate a linked list consisting of each of the drive types. This is anchored by a global pointer, and accessible to all parts of the program. Thus when a section of the program requires to list the drive types it runs through the drive list and writes out the type of drive from that field in the definition of the drive type. The code for the function drivelist is given below :

```
drivellst()
{
  Drivenone * gen = new Drivenone;
  strncpy(gen->namestr,"gen",20);
  gen->nextptr= NULL;
  Drivesiren * siren = new Drivesiren;
  strncpy(siren->namestr,"siren",20);
  siren->nextptr= NULL;
  Drivetimer * timer = new Drivetimer;
  strncpy(timer->namestr,"timer",20);
  timer->nextptr= NULL;
  drivellst = gen;
  gen->nextptr= siren;
  siren->nextptr= timer;
}
```

This is the only section of the program where the code is hard wired to class definition. If the program was sold as a commercial entity, with all the base classes already compiled, then the code for this procedure would have to be given and compiled into the user's code.

The advantages of using this approach will be demonstrated later where by virtue of the fact that a pointer of the base class type is pointing to an item in the above list, a new variable of that type can be created by using the member function add to list.

The second function to be considered is called add drive. This is used to add drives to the linked list of all drives. This is a generic procedure that is called globally from the main operating menu. The code segment is given below :

```
adddrive(GeneralDrive * current)
{
char names[20];
int  select,lines;
 while (current != NULL)
   {
        list = current;
        current = current->nextptr;
   }
      current = list;
         do
            {
             lines = 1;
             printf( "%s\n","   DRIVE TYPES ");
            list = drivelist;
             while (list != NULL)
                   {
                    printf("%d%s",list->type," )  ");
                    printf("%s\n",list->typename);
                    list = list->nextptr;
                    lines ++;
                   }
              printf("%s"," Choose a drive type (0 for none) : ");
              scanf("%d",&select);
             if ((select > 0) &(select < lines ))
               {
                printf("What is the drive name : ");
                scanf("%s",names);
                  list = drivelist;
                  lines = 1;
                    while ((list != NULL) & (lines < select))
                      {
                        list = list->nextptr;
                        lines++;

                      }
                    if ((select > 0) & (list != NULL))
                    {
                    list->addtolist(names);
                    current->nextptr = temp;
                    current = temp;
                    }
               }
             if (lines > select)
               {
                  printf("%s\n","PLEASE TRY AGAIN");
               }
            } while (select > 0);
}
```

The section from the first assignment "list = drivelist" to the following if statement lists the various drive types from the global list as described previously. once the user has selected one the second code segments starting from the same assignment statement searches the list and points a pointer (of class general drive) the selected type of drive. This shows the power of the design methodology where a pointer of one type does not have to have its type changed when scanning the list.

The statement "list->addtolist(names)" really embraces the power of Object-oriented programming. This calls the class to which it is currently pointing and passes it a message which tells it to create a new drive of that class and assign the variable "name" to its variable called "name str". The assignment immediately following this takes the current pointer (pointing to the last drive in the list) and points it to the newly created drive. Thus there is no code in this segment that is dependant on the number of subclasses of general drive that exist. In fact the only place where this is needed is in the drive type list discussed previously.

A similar methodology is employed in the function to read the stored data from the storage medium. (function f add) This will be discussed later as there is substantially more code in this function to deal with the rest of the data that is stored.

The list of operating drives (as opposed to the list of drive types) could have easily been created as a class of its own. However there is only one occurrence of this list type and therefore it was decided to use a more conventional list.

A corresponding procedure also exists to delete a drive form the list. The code segment for this procedure is given below :

```
deletedrive()
{
  int select;
  GeneralDrive * temp;
  listdrives();
  current = first;
  printf("\n Which drive do you want to delete : ");
  scanf("%d",&select);
  current = finddrives(select);
  temp = current->nextptr;
  current->nextptr=current->nextptr->nextptr;
  delete temp;
}
```

The operation of the function is fairly straightforward. It lists all the drives previously created (Function listdrives) and asks for a selection. The function finddrives searches the list of drives and points to the one before the selected drive. This is to allow the next pointer from that drive to be pointed over the selected drive and the space occupied by the drive to be freed. This is accomplished by the delete (standard C++) function. The code segments for the functions described is listed below.

```
GeneralDrive * finddrives (int i)
{
int j;
j=0;
do
{
current = current->nextptr;
j++;
} while ((j!=i-1)&(current->nextptr!=NULL));
return(current);
}
```

The above function is of a pointer type and returns a pointer to the drive before the one selected.

```
int listdrives ()
{
int i;
i=1;
current=current->nextptr;
do
{
cout << i << ".) " << current->namestr << "      "<< current->typename << "\n";
current = current->nextptr;
i++;
} while (current!=NULL);
return(i-1);
}
```

The function list drives returns an integer value of the number of drives listed. This is useful in ranging selections that have to be made i.e. when the user has to select one of the drives listed. It can be seen that all the program segments will contain range checking after a selection. The upper limit of this range is determined by the result of the function.

There are a few more global procedures that are used but these relate to the other independent classes and will be discussed in the context of these. It must be noted that the global list management could have been done by means of a

new class called list manager. However, there would only be one instance of this class and therefore it was decided to leave these as global procedures.

6.3 Interlocks

The purpose of interlocks in a plant are to ensure that certain operations occur that have a dependance on others. Simply put this means that a drive may not start unless certain conditions are met. These are not related to the actual drive IO but rather to the state of other pieces of equipment in the plant.

The interlock was mentioned in the previous chapter in regard to a the structure of a general drive having an interlock as one of its variables. This section will explain the derivation of the class interlock and the support infrastructure necessary to make interlocks part of the control system.

An interlock generally consists of a number of conditions and not merely a one to one relationship between the current drive and another piece of equipment. The list of interlocks is often very large (see chapter 7) and varies in size from drive to drive.

Given the above description it was decided to implement interlocks as a linked list of interlock nodes. The variable associated with each drive is of the class interlock node. This has a pointer to the first of the nodes in the list. Each node consists of a pointer to the name of the drive to which it is pointing, as well as

a point to the running status of that drive. This does not necessarily have to be a running signal, it could be a high level alarm or another digital signal.

The definition of the interlock classes is given below :

```
class interlocknode
{
public:
    interlocknode * next;
    int * runstate;
    char name[20];
};

class interlock
{
  public:
        interlocknode  * firstnode;
        interlock();
        list(int * nocflines);
        int setstatus();
};
```

The class interlock has two member functions the first of which is called list. This is used to list all the drives that form part of the linked list of the interlock. This is necessary in order to be able to modify these interlocks. The second is an integer function that calculates the status of the list of interlocked drives. It checks that all the interlock signals are healthy and if so returns the value 1. This value is used in the operate1 function of the General drive definition (chapter 6). If the state of this value is not 1 then the drive is not permitted to start. The following code contains the definitions for the two classes listed above.

```
interlock::setstatus()
{
    interlocknode * currentnode;
    int i;
    i = 1;
    currentnode = firstnode->next;
       while (currentnode != NULL)
         {
```

```
            l =((l) & (* currentnode->runstate));
            printf(" Interlock state %d",l);
            printf(" after drive  %s\n ,currentnode->name);
            printf("\n");
            currentnode = currentnode->next;
          }
     return(l);
}
interlock::list(int * nooflines)
{
    interlocknode * currentnode;
    int i;
    i=1;
    currentnode = firstnode->next ;
 do
  {
    printf ("  %d", i);
    printf ("%s",".) ");
    printf ("     %s\n",currentnode->name);
    currentnode = currentnode->next;
    i++;
  } while (currentnode!=NULL);
*nooflines = i-1;
}


interlock::interlock()
 {
  firstnode = new interlocknode ;
  firstnode->next=NULL;
  strcpy(firstnode->name,"first");
  *firstnode->runstate = 1;
 }
```

In addition to the above definitions there are global procedures used to add, delete and modify interlocks. However the actual code for adding and deleting interlocks is part of the definition of the general drive. This provides a safety feature in that the list of interlocks is an integral part of the drive and not an independent entity. The code below describes the three procedures :

```
addinterlock()
{
  GeneralDrive * thisone;
  int select,no;
 do
 {
  current = first;
  no = listdrives();
  printf("\n To which drive do you want to add the interlocks ( 0 to quit): ");
```

```
  scanf("%d",&select);
printf("debug  select : %d\n",select);
  if ((select > 0) & (select < no))
      {
          current = first;
          thisone = finddrives(select)->nextptr;
          thisone->addinterlocks();
      }
      printf(" Drive name : %s\n",thisone->namestr);
      thisone->interlock.list(&no);
 } while (select > 0);
}

delinterlock()
{
  GeneralDrive * thisone;
  int select,no;

 do
 {
  current = first;
  no = listdrives();
  printf("\n From which drive do you want to delete an interlock ( 0 to quit): ");
  scanf("%d",&select);
  if ((select > 0) & (select < no))
      {
          current = first;
          thisone = finddrives(select)->nextptr;
          thisone->delinterlocks();
      }
  thisone->interlock.list(&no);
 } while (select > 0);
}
modinterlock()
{
  GeneralDrive * thisone;
  int select,no;

  current = first;
  no = listdrives();
  printf("\n Which drive's interlocks do you want to modify ( 0 to quit): ");
  scanf("%d",&select);
  if ((select > 0) & (select < no))
      {
          current = first;
          thisone = finddrives(select)->nextptr;
     do
       {
         printf(" Options : \n");
         printf(" 1) Add a drive to the interlock : \n");
         printf(" 2) Delete a drive from the interlock : \n");
         printf(" 3) Quit : \n");
         printf(" selection : ");
         scanf("%d",&select);
         switch(select)
         {
           case(1):thisone->addinterlocks();
```

```
                break;
            case(2):thisone->delinterlocks();
                break;
            default:break;
          }
        } while (select < 3);
      }
}
```

In essenc ail these procedures do is to select the drive whose interlocks need to be modified and then calls that drive and it executes its own modification. Once again this uses the power of the Object-Oriented structures to allow fairly simple global procedures that call the more complex member functions of a particular class.

6.4 Sequences

As the trend towards automation increases there is a growing use of sequencing in industrial plants. This in effect automates the sequence of events necessary to control a section of a plant and replaces the need for an operator to start a set of drives in a manual sequence. There is an obvious need for increased safety in this regard as the control system must obey the rules related to the correct operating conditions for each drive.

A sequence merely carries out a set of logical steps to start or stop a set of drives in a predefined order. The actual definition of a sequence is fairly similar to that of an interlock. However, the sequences are independent of any particular

drive and thus need to be managed in a slightly different manner to the interlocks.

Each sequence consists of a base node of the class sequence and a series of linked nodes called seq nodes. All the defined sequences are formed into a linked list which is held by a global anchor called first sequence. There are more global procedures related to sequences than for interlocks. The reason is the relative independence of the sequences from the actual drives. Thus all the manipulation procedures are independent and a drive that is part of a sequence has no indication thereof. The class definitions follow :

```
class sequencenode
 {
public:
    sequencenode * next;
    sequencenode * previous;
    GeneralDrive * drive;
};

class sequence
{
  public:
        char name[10];
        sequencenode  * firstnode;
        sequence * nextseq;
        sequence();
        setup();
};
```

The difference between this list and a the other lists is that each node has both a backward and forward pointer. The sequence itself has a pointer to its first node as well as a pointer to the next sequence in the list of sequences. The function setup() is used to create a sequence and is thus part of the class definition thereof. Global procedures used in sequence management are listed below :

```
int listsequence (sequencenode * firstnode)
{
    sequencenode * currentnode;
    int l;
    l=1;
    currentnode = firstnode->next;
 do
  {
    printf ("  %d", l);
    printf ("%s",".) ");
    printf ("%s",currentnode->drive->namestr);
    printf ("%s\n",currentnode->drive->typename);
    currentnode = currentnode->next;
    l++;
  } while (currentnode!=NULL);
   return(l-1);
}
```

The list sequence procedure is used to list the contents of a sequence i.e. the drives that make up the sequence. The procedure that follows (list sequences) lists all the sequences and then calls the above procedure to list the individual drives in the sequence.

```
listsequences( int * nooflines)
{
  int l,dummy;
  l = 1;
  currentsequence = firstsequence->nextseq;
  while (currentsequence != NULL)
    {  printf("%d",l,"%s"," ).  ");
       printf("%s\n",currentsequence->name);
       dummy=listsequence(currentsequence->firstnode);
       currentsequence = currentsequence->nextseq;
       l++;
    }
    *nooflines = l-1;
}

addsequence()
{
  int select;
  currentsequence = firstsequence;
  while (currentsequence->nextseq != NULL)
    { currentsequence = currentsequence->nextseq; }
 do
   {
    select = 1;
```

```
   if (select == 1)
      {
       printf(" What is the sequence name : ");
       currentsequence->nextseq = new sequence;
       scanf("%s",currentsequence->nextseq->name);
       currentsequence = currentsequence->nextseq;
       currentsequence->setup();
      }
    printf(" Do you want another sequence 1 for yes : ");
    scanf("%d",&select);
    } while (select == 1);
}

delsequence()
{
  int select,i,count;
  sequence * temp;
  sequencenode * currentnode;
  sequencenode * tempnode;
  count = 1;
  listsequences(&i);
  printf(" Select a sequence no : (0 for none) ");
  scanf("%d",&select);
   currentsequence = firstsequence;
    if ((select > 0) & (select < i))
     {
      while ((count < i) & (count <select))
        {  currentsequence = currentsequence->nextseq;
           count++;
        }
          temp = currentsequence->nextseq;
          currentsequence->nextseq =
                          currentsequence->nextseq->nextseq;
          currentnode = temp->firstnode->next;
          tempnode = currentnode;
          while   (currentnode !=NULL)
              { free(tempnode);
                tempnode = currentnode;
                currentnode = currentnode->next;
               }
               free(currentnode);
     }
    listsequences(&i);
}

modsequence()
{
  int select,i,count;
  sequence * temp;
  sequencenode * currentnode;
  sequencenode * tempnode;
  count = 1;
  listsequences(&i);
  printf(" Select a sequence no : (0 for none) ");
  scanf("%d",&select);
   currentsequence = firsts  quence;
    if ((select > 0) & (select < i))
```

```
{
 while ((count < l) & (count <select))
   {  currentsequence = currentsequence->nextseq;
      count++;
   }
     temp = currentsequence->nextseq;
  do
   {
     printf(" Options : \n");
     printf(" 1) Add a drive to the sequence : \n");
     printf(" 2) Delete a drive from the sequence : \n");
     printf(" 3) Quit : \n");
     printf(" selection : ");
     scanf("%d",&select);
     switch(select)
     {
       case(1):temp->setup();
               break;
       case(2):count=listsequence(temp->firstnode);
               printf(" Select a drive (0 for none)  :  ");
               scanf("%d",&l);
                 if ((l > 0)&(l<count+1))
                   {
                     count = 1;
                     tempnode=temp->firstnode->next;
                     currentnode = temp->firstnode;
                       while (count < l)
                          {
                            currentnode = tempnode;
                            tempnode = tempnode->next;
                            count++;
                          }

                       currentnode->next=tempnode->next;
                       free(tempnode);
                   }
               break;
       default:break;
     }
   } while (select < 3);
 }
}
```

A number of the above procedures could perhaps have been defined as member functions, but it was decided to rather define them globally. The rationale was that they are genera' enough to be used for any form of list management.

The last sequence related function is used to start a sequence of drives. This is again a global procedure as if it were a member function there would still have to be a global procedure that called it.

The code segment for this procedure is given below :

```
operateseq()
{
  int nooflines,select,i;
  sequencenode * currentnode;
  do
    {
      listsequences(&nooflines);
      printf(" select a sequence (0 for none : ");
      scanf("%d",&select);
        if ((select > 0) & (select < nooflines + 1))
          {    i = 1;
               currentsequence = firstsequence->nextseq;
               while ((currentsequence != NULL) & (i < select))
                  {
                     currentsequence = currentsequence->nextseq;
                     i++;
                  }
               currentnode = currentsequence->firstnode->next;
                  while (currentnode != NULL)
                     {

                    currentnode->drive->rack.seqstart.setDI(1);
                        currentnode->drive->operate(0,0);
                       currentnode->drive->status();
                       currentnode = currentnode->next;
                     }
          }
    } while (select > 0);
}
```

The heart of the procedure sets a digital value called seq start and then passes the message operate to the drive to which it is currently pointing. This is to allow the drive to start if its trips and other conditions are healthy. The procedure then passes the message status to the drive. The drive then lists the status of its IO rack and other digital values. This is done in turn for each of the drives in the

sequence. When the message operate is passed to the drive, it checks its interlocks and then starts itself. This again demonstrates the differences between Object-oriented design methods and conventional software design methodology.

6.5 Menu Management

There are a few procedures and global variables that are defined to aid in the operation of the software. These form the heart of the menu management system. All the text for the menus are defines globally as :

```
static char * mainmenu[] = {
        "       OPTIONS MENU      ",
        "1)  ADD A NEW DRIVE  ",
        "2)  REMOVE A DRIVE   ",
        "3)  SET DRIVE INTERLOCKS ",
        "4)  DELETE DRIVE INTERLOCKS ",
        "5)  MODIFY DRIVE INTERLOCKS ",
        "6)  OPERATE A DRIVE ",
        "7)  RE - LOAD ALL INFO",
        "8)  CREATE A SEQUENCE",
        "9)  DELETE A SEQUENCE",
        "10) MODIFY A SEQUENCE",
        "11) OPERATE A SEQUENCE",
        "12) SAVE ALL INFO",
        "13) SHOW ALL DRIVES",
        "14) LIST SEQUENCES",
        "15) QUIT",
        "LAST"
        };

static char * operate[] = {
        "  Operate Menu ",
        "0) DO NOTHING",
        "1) FIELD START THE DRIVE ",
        "2) TRIPS HEALTHY - CHANGE STATE",
        "3) STOPS HEALTHY - CHANGE STATE",
        "LAST"
        };
```

These static character arrays are called from an integer function called menu. This function lists the options returns a response value to the part of the program that called it in the first place.

The menu function provides a simple way of managing the operation of menus without cluttering up the code associated with the choice.

```
int menu(char * lines[], int * nooflines)
{
  int select,i;
    i=0;
    while (strcmp(lines[i],"LAST")!=0)
       {
        printf("%s",lines[i]);
        printf("\n");
        i++;
       }
    i=i-1;
    *nooflines = i;
    printf("MAKE A SELECTION AND HIT ENTER : ");
    scanf("%d",&select);
    return(select);
}
```

The main body of the program is controlled by a procedure called firstmen. (first menu) This contains a case statement contained in a while loop. All the main functions ir the program are executed from this menu system.

```
firstmenu()
{
 int select=0;
 int nooflines=0;
 int dummy,i;
 do
  {
   select = menu(mainmenu, &nooflines);
  printf(" debug select %d\n",select);
  current = first;
    currentsequence = firstsequence;
    switch(select)
     {
    case(1):adddrive(current);
            break;
    case(2):deletedrive();
```

```
            break;
      case(3):addinterlock();
            break;
      case(4):delinterlock();
            break;
      case(5):modinterlock();
            break;
      case(6):operatedrive();
            break;
      case(7):fadd();
            break;
      case(8):addsequence();
            break;
      case(9):delsequence();
            break;
      case(10):modsequence();
            break;
      case(11):operateseq();
            break;
      case(12):fsave();
             break;
      case(13):listdrives();
               printf("strike any key when ready");
               dummy = getch();
               break;
      case(14):listsequences(&l);
            printf("strike any key when ready");
            dummy = getch();
            break;
       default:break;
       }
     } while (select < nooflines);
}
```

6.6 File Storage and Retrieval

The storage and retrieval of data from the disc is controlled by two procedures viz f add and f save. These two merely retrieve and save data in a specified format. They both default to a data file called "data.txt" but allow the user the option of changing the name of the file. The code for these procedures follows :

```
fadd()
{

char names[20];
int  select,lines,options;
```

```
FILE *fin;
char fname[11];
 printf(" Do you want to change th  file name (1 for yes) : ");
 scanf("%d",&select);
 if (select != 1)
   {
    fin = fopen("data.txt","r");
   }
  else
    { printf (" What is the file name : ");
      scanf("%s",fname);
      fin = fopen(fname,"r");
    }
 if (!fin)
  {printf(" error in file ");}

// reads drives

printf(" Reading drives \n");
    current = first;
          list = drivelist;
           options = 0;
            while (list != NULL)
              {
               list = list->nextptr;
               options ++;
              }
    fscanf(fin,"%d",&select);
     if (select > 0)
     {
        do
         {
               fscanf(fin,"%s",names);
printf(" Reading : %s\n",names);
               list = drivelist;
               while ((list != NULL) & (list->type != select))
                  {
                     list = list->nextptr;
                  }
               if (list != NULL)
                  {
                     list->addtolist(names);
                     current->nextptr = temp;
                     current = temp;
                  }
            fscanf(fin,"%d",&select);
         } while (select > 0);
     }

// reads sequences

printf(" Reading sequences \n");
sequencenode * currentnode;
char seqname[10];
char drlname[20];
    currentsequence = firstsequence;
    fscanf(fin,"%s",seqname);
```

```
printf(" Reading : %s\n",seqname);
 if (strcmp(seqname,"zz")==0)
  {
    fscanf(fin,"%s",seqname);
printf(" Reading : %s\n",seqname);
    while (strcmp(seqname,"zzzz")!=0)
      {
        currentsequence->nextseq = new sequence;
        strcpy(currentsequence->nextseq->name,seqname);
         fscanf(fin,"%s",driname);
printf(" Reading : %s\n",driname);
         currentnode = currentsequence->nextseq->firstnode;
        while (strcmp(driname,"zz")!=0)
         {
           current = first;
           while (strcmp(driname,current->nextptr->namestr)!=0)
             {
             current = current->nextptr;
             }
               sequencenode * nextnode = new sequencenode;
               currentnode->next = nextnode;
               nextnode->drive = current->nextptr;
               nextnode->next=NULL;
               nextnode->previous = currentnode;
               currentnode = nextnode;
               fscanf(fin,"%s",driname);
printf(" Reading : %s\n",driname);
          }
           fscanf(fin,"%s",seqname);
printf(" Reading : %s\n",seqname);
          currentsequence= currentsequence->nextseq;
      }

  }
char drname[20];
GeneralDrive * tmpdrive;
interlocknode * currentnod;
printf(" Reading Interlocks \n");

  fscanf(fin,"%s",drname);
  printf(" Reading : %s\n",drname);
  if (strcmp(drname,"xx")==0)
   {
    fscanf(fin,"%s",drname);
  printf(" Reading : %s\n",drname);
    while (strcmp(drname,"xxxx")!=0)
      {
       current = first->nextptr;
          while(strcmp(drname,current->namestr)!=0)
            {
              current = current->nextptr;
            }
       tmpdrive = current;
       currentnod = tmpdrive->interlock.firstnode;
        do
          {
            fscanf(fin,"%s",drname);
```

```
  printf(" Reading : %s\n",drname);
              current = first->nextptr;
              if (strcmp(drname,"xx")!=0)
               {
                   while(strcmp(drname,current->namestr)!=0)
                     {
                       current = current->nextptr;
                     }
                  interlocknode * nextnod = new interlocknode;
                  currentnod->next = nextnod;
                  (nextnod->runstate) =&(current->rack.start.state);
                  strcpy(nextnod->name,current->namestr);
                  nextnod->next=NULL;
                  currentnod = nextnod;
               }
            } while (strcmp(drname,"xx")!=0);
          fscanf(fin,"%s",drname);
   printf(" Reading : %s\n",drname);
         }
     }
fclose(fin);
}



fsave()
{
FILE *fout;
int select;
char fname[11];
 printf(" Do you want to change the file name (1 for yes) : ");
 scanf("%d",&select);
 if (select != 1)
   { fout = fopen("data.txt","w");  }
  else
    { printf (" What is the file name : ");
      scanf("%s",fname);
      fout = fopen(fname,"w");
    }
 if (!fout)
  {printf(" error in file ");}
// saves all drives


current=first->nextptr;
while (current!=NULL)
   {
    fprintf(fout,"%d\n",current->type);
    fprintf(fout,"%s\n",current->namestr);
    printf(" saving %s\n",current->namestr);
    current =current->nextptr;
   }
       fprintf(fout,"%d\n",0);
       fprintf(fout,"%s\n","zz");
```

```
// saves all sequences
 sequencenode * currentnode;
 currentsequence = firstsequence->nextseq;
  while (currentsequence != NULL)
    {  printf(" saving ");
       printf("%s\n",currentsequence->name);
       fprintf(fout,"%s\n",currentsequence->name);
       currentnode = currentsequence->firstnode->next;
         do
          {
            fprintf(fout,"%s\n",currentnode->drive->namestr);
            currentnode = currentnode->next;
          } while (currentnode!=NULL);
          fprintf(fout,"%s\n","zz");
          currentsequence = currentsequence->nextseq;
    }
  fprintf(fout,"%s\n","zzzz");


// Saves Interlocks

  current=first;
  interlocknode * currentnod;
   while (current!=NULL)
     {
       if (current->interlock.firstnode->next!=NULL)
            {
              currentnod = current->interlock.firstnode->next;
                 fprintf(fout,"%s\n","xx");
                 fprintf(fout,"%s\n",current->namestr);
                 while (currentnod != NULL)
                    {
                      printf(" saving interlock: %s\n",currentnod->name);
                      fprintf(fout,"%s\n",currentnod->name);
                      currentnod = currentnod->next;
                     }
            }
         current = current->nextptr;
     }
  fprintf(fout,"%s\n","xx");
  fprintf(fout,"%s\n","xxxx");
  fclose(fout);
}
```

6.7 Conclusion

It can be seen that there is a large amount of infrastructural software that has been developed. An attempt has been made to embrace the principles of Object-Oriented design even in the peripheral software. If is evident from the previous sections that this has been possible to a large extent. It has also shown the power of this design methodology as an alternative form of software design

It is a great pity that technology has not allowed for the use of languages such as C (not even Object-oriented C) in the field of PLC programming. It is believed that this will one day be the case. at which point the principles discussed in the previous chapters will come to the fore as a major force in the control system field.

7.0 SOFTWARE FOR CRUSHING AND SCREENING PLANT

7.1 Introduction

This chapter considers an application of Object-oriented design techniques as applied to a real world problem. The area to be considered the design of the software for the Primary crushing and screening section of a diamond processing plant.(Ireland 1987) The circuit is fairly simple in operation and is the primary source of ground being fed into the plant. This will be considered from a number of viewpoints :

The application of Object-oriented techniques

The development of different classes and subclasses

The effectiveness of the application of these design techniques

These are just a few of the areas to be covered in the design of the software.

7.2 Functional Description

The following is an extract from the functional description as written by Ireland (1987). This is only a small portion of the description of the plant.

"Run of mine ore is tipped through a 800mm aperture static grizzly into a 150 ton receiving bin. Any oversize material is broken up by a rockbreaker at the bin. A pushfeeder (No P1) then feeds the material onto a vibrating screen (No S1) screening at 200mm. The -200mm

Material is fed onto no 1 conveyor via a chute. From no 1 conveyor the material is delivered onto two screens (Nos S2 and S3) screening at 50mm. The oversize passes onto conveyor 5 and from there is transported onto the boulder stockpile. The -50mm undersize falls into the primary screening bin.

The +200 oversize reports to a jaw crusher (J1) where it is crushed to -125mm. From the crusher the material is fed onto a flats removal screen (S4). The -200mm material is screened off onto no 2 conveyor which feeds the primary screening bin. The oversize material is fed to the waste stockpile via conveyor 4. A pushfeeder under the primary screening bin loads conveyor 3 which transfers material to the scrubber bins."

7.3 Start-up Sequence

The screening section must be started up before the crushing section as the latter feeds the former. The screening section must be started up as follows:

start siren 2

wait 6 seconds

stop siren

wait 14 seconds

start conveyor no 50

start screen no 15

start screen no 16

Once the above section has been started in the correct order then the primary crushing section can be started in the following order:

Start siren 1

wait 6 seconds

stop siren 1

wait 14 seconds

start conveyor 1

start conveyor 12

start slab screen 42

start conveyor 2

start jaw crusher lube system

start jaw crusher

start screen no 35

start no 1 pushfeeder hydraulic pump

start no 1 pushfeeder

set rockbreaker cooling pump on

Both of the above circuits will be shut down in the reverse order to the start-up.

7.4 Interlocking

Jaw crusher J1

This drive is interlocked with the jaw crusher pressure switch and the liquid starter

Pushfeeder no 1 P1 main motor

This drive is interlocked with the following:

Hydraulic motor no 1, Screen 1, screen 2, Conveyors 1, 2 and 4, Primary screening bin high level, primary tipping bin low level, Boulder stockpile high level and slabs stockpile high level.

The number one pushfeeder will only start if the above drives are running and if the level alarms given above are in a non-alarm state.

Screen 1 S1

This is interlocked with the jaw crusher, slab screen and conveyor no 1.

Screen 4 S4

This is interlocked with conveyors 2 and 4.

Conveyor 1

This conveyor is interlocked with screens 2 and 3 as well as its discharge chute blocked chute detector. In addition this conveyor has a tensioning winch attached to it. This winch will pretension the belt so as to allow for smooth operation. This winch is reversible and has trip and stop signals interlocked to it. The winch must start and tension the belt before the belt drives are started.

Conveyor 5

This conveyor is only interlocked with the boulder stockpile high level.

Other drives

The following drives are only interlocked to safe start sirens

Screens 2 and 3(S2 & S3), Conveyors 2 and 4 and no 1 pushfeeder hydraulic pump.

Each of the drives listed is also interlocked with its own trip and stop signals. This implies that if the drive has tripped or the stop button has been depressed then the drive will not start.

The majority of the interlocks listed are concerned with safety features of the system. This ensures that if a drive does stop for any reason then any other equipment that is dependent on it will also stop.

It must be remembered that the description given only covers a small portion of the plant and therefore will serve as a basis for developing the software that would be used for the entire plant.

7.5 Software design

In chapter three a generalized method of software development was discussed. This method embraces the concepts of object oriented design in their entirety. The development of software for the section of the diamond processing plant discussed previously will be considered using the method discussed.

The problem has been defined in terms of the plain language definition of the problem. Here a full description of the system was given and included a description of all the items of relevant equipment and a definition for the basis of their interaction.

The phase involving the informal strategy has to some extent been defined. Following on from the functional description of the plant a description of the interlocking and starting sequences has been given. The assumption has been made and is generally made that the basic mode of operation of all the equipment is the same. This means that the basic unit in the plant will have trips-healthy and stops-healthy signals which must be taken into account. It is at this stage that the mode of operation of the generalized drive is formulated. In the case of PLC program development the CSF method of program description is often used. A description of the mechanics of the operation is given in chapter two. A PDL overview of the entire operation could be utilized at this stage for the description of complex interaction between the different units. In the case of the diamond plant however the generalized description the unit operation will suffice.

The formalization of the strategy involves the identification of the objects and hence the different classes of objects. In the example used the "objects" are the drives used for the conveying of material. In this instance and most applications of this type there will be very few if any abstract data types or objects. There are differences in the actual operation of the individual drives which were described in the section under interlocking. At this point certain aspects of each drive were also discussed. These include features such as reversibility of drives, and tensioning winches on long conveyor belts. The objects identified are as follows:

"Screen"

"Uni-directional_conveyor"

"Jaw_crusher"

"bi-directional_winch

"standard_pump

"push_feeder"

"level_detector"

There are similarities in all the different types of drive operations. It is possible therefore to utilize the class "General_drive" as defined in chapter 5. This would form the basis from which to work when defining the other objects and facilitate the subclassing.

The basic superclass of "General_drive" can now be subclassed into the types of objects discussed above. Whilst still in the formalization of strategy phase we

should also note that the operation of "Uni-directional_conveyor", "screen" and "Push_feeder" are all similar in operation. Therefore a class called "Drive_with interlocks" could be established. This also has a minor change from the General drive. In this case there is not a field start but rather an operator start. This signal is sent from the control room as opposed to a push-button in the field. In the past this signal originated from a push-button on the operator monitoring console. Today drives within an industrial plant are controlled from a supervisory computer. This also forms a database function and has graphic displays for easy reference to plant conditions.

The next object to consider is the "Jaw_crusher". This is two drives in one, with the jaw crusher and its associated lube pump. Here the two drives are seemingly separate except for the case when the crusher motor cannot start without the lube-pump running and must stop 30 minutes after the lube pump in the event of a trip.

Another object that consists of two drives in one is the "Conveyor_with winch". This may be broken down into two classes of object each consisting of one of the drives. However if multiple instances of the same situation i.e. a tensioned conveyor, exist then it is perhaps best to establish a class of its own.

The class of "standard_pump" has been considered numerous times within the course of this dissertation and is not discussed as it is merely a General_drive.

At this point the entire area of formalization of the strategy has been carried out up to the stage of implementation of the objects into specific modules in a language. Once the objects have been implemented there is one more step before the testing phase. This step involves returning to the functional description of the process. One section of this description involved the sequence order of starting the drives. This forms one of the abstract objects that are used in the control of plants of this nature.

The previous chapters defined the sequences and interlocks that are used in industrial plants. These could be applied as they are to the example given in this chapter.

7.6 Conclusion

In principle the basic object oriented techniques can be applied to the development of control system software. This methodology allows for more efficient software development as the basic logic has been defined, the programmer merely needs to define the variables and any additional classes required. The majority of the new classes are derivations of the previous classes.

It can be seen that all the principles discussed within the scope of this dissertation provide for the fundamental changes required in the design of the control systems for industrial plants. This chapter showed on a conceptual level

how a fairly intensive infrastructure can be applied to the simple development of rather complex control systems.

An issue that has not been addressed was that of analogue signals. These are generally required as indications, or inputs to PID control loops within a plant. The principles discussed can equally be applied to both areas of analogue value usage. In fact the class of PID controller provides another useful application of Object-Oriented design. This is however impractical in this instance due to the level of emulation required in demonstrating the principles.

8.0 CONCLUSION

The basic hypothesis of this dissertation was to demonstrate that the principles of Object-Oriented design can be applied to the development of control system software. In addition, the objective was to show why object oriented design should be used, as opposed to an alternative development paradigm. It is believed that the objective was met and the principles have been demonstrated. These are now discussed in an attempt to justify the previous statement.

The basic four tenets of object oriented design, as discussed in chapter three, define the features of languages, that will allow those languages to be defined as object oriented. However, this dissertation began from the basic premise that C++ is an object oriented language and that these four facets would form the demonstration of the basic hypotheses. Thus, the application of these features will form the underlying demonstration that the techniques of object oriented design can be applied to the design and implementation of control system software.

The first of these is the principle of data abstraction. This was used extensively in the definition of the new classes, from the base class definition. This formed one of the bases for the development of the rest of the software. It allowed a basic class to be defined, and additional classes that shared the same basic behavior to be subclassed from this basic class. This has a substantial impact on the programming required, and implications for software re-usability. In the

former case, the subclass definitions required far less programming than if each was to be generated as an independent class. The worst case would be if each had to be continually defined each time it was required, as is the case with the majority of PLC systems. The latter aspect, of software re-usability, can be applied where a number of basic class definitions can serve as the basis for a large number of different programs. Thus the programmer's task is greatly reduced.

Information hiding has been applied in the safety aspects of control systems. This was shown in chapter 5 in the definitions of the I/O. the ideal situation would be that all I/O is defined at the start of the program and assigned to drives. After that point, no other drive would be able to access the I/O from another drive. The second area where this was used, was in the definitions of local functions and procedures. This ensured that an object, not of that class, did not have access to that function.

The principles of dynamic binding came to the fore in the drive definitions. Here each subclass had a few of the member functions declared, in the superclass and the subclass. This concept of operator overloading allowed the different objects to behave in fundamentally different ways to the same message being passed to them. This is perhaps one of the most powerful of the features of object oriented programming languages. It was used in a number of instances, but two will be used for the purposes of example. In the section of code to add drives to the linked list, a pointer points to a drive of the class to be added and

then the message is passed to the class to add itself to the list. This requires that each drive allocate a different amount of memory to itself and initialize all its members. This proved to be tremendously useful and reduced the amount of hardwired code required. This will be discussed later in the conclusion. The second example is in the sequencing, where when starting a sequence of drives, the message is passed to the drive and it operates itself. This is independent of the drive type as the program decides which operate function to call.

The final area, inheritance, has been covered to some degree. In this arena, each subclass inherits the functions and characteristics of the superclass, and adds some of its own features. This is used extensively in the definitions of the drive classes.

There are a number of other features of object oriented design that need to be considered. The linked list used to combine all the drives into one list was not homogeneous, but rather a heterogenous list. It was composed of fundamentally different classes, whose only commonality was their base of derivation. This is obviously an important aspect, as it allows for considerably easier management of the data within the program infrastructure.

On the PLC side, each time a drive is called, the same mnemonics are used i.e. start, trips healthy etc. The actual I/O addresses are hidden. This leads to simplification as for each drive the signals related to that drive are all called the same name.

The last aspect that is considered is the ability to define totally abstract classes. The drive classes each refer to a piece of physical equipment, whereas sequences and interlocks are abstract concepts. This definition ability is fundamental in developing software of this nature.

The emergence of the newer Object-oriented languages has made the implementation of such software a far easier task than previously. The early editions of Smalltalk, the first true Object-Oriented language, featured an extensive background infrastructure that required the grasping of a totally new programming environment. The language used for the current implementation (Zortech C++) as well as other versions of C++ allow a programmer a more conventional type of compiler based programming infrastructure. This still retains the basic principles of the design methodology and the power of Object-Oriented design.

It is believed that the extent to which the features of object oriented design have been applied within the scope of this dissertation refutes the claims of using another software development paradigm. In fact, only languages that display the four basic characteristics of object oriented languages would allow for development of this nature. Thus the second major hypothesis of this dissertation has been demonstrated in that object oriented design should be used in the development of software, specifically of this nature, in the place of other software development paradigms. It must be stressed that it is difficult to develop a

comparison between the object oriented approach and current PLC software development paradigms. The reason is that the latter do not really exist. PLC software development paradigms are not really well defined. It appears as if each programmer has developed their own paradigm, based upon one of the more classical software development paradigms. However, PLC software design does not follow one of the traditional methodologies. Thus the comparison is not really valid, as there is not one set methodology.

An interesting development over the last year is that a number of compilers for Object-Oriented languages feature an intermediate stage of compilation that allows the generation of a standard form of a conventional language. Thus the principles of the methodology are applied, but can be listed in a language standard form. This obviously defeats some of the objective, but allows for a set of standards to be created.

At the start of this dissertation it was commented that there are a number of authors who discuss the downfalls of the languages as opposed to their application. A comment must be made as to the state of PLC languages and that it is hoped that they will be brought up to the current levels of technology in the future. The programming on C++ has provided an opportunity to explore this exciting new methodology of software design. It was decided at the outset to attempt to apply this methodology to a real world problem. This was perhaps the fundamental flaw in a number of the papers on the subject. They only tended to address the fairly esoteric issues and not a fundamental real live problem.

In conclusion, I believe that I was swept up the wave of optimism that surrounded the beginnings of object oriented design. However, as time has progressed I have seen object oriented design as an extension to our basic software design tools, and not as the universal panacea. I see object oriented design as addressing some of the more conventional problems of software design and structure. It is believed that the approaches adopted meet both the academic requirements of the Master of Science in Engineering as well as being able to provide an insight into the use of Object oriented design as a practical tool.

REFERENCES AND BIBLIOGRAPHY

ABBOTT R. 1980 "Report on teaching ADA", Science applications INC, Rep SAI-81-312WA, December 1980.

ADLEMAN M.L.,1987 " Programmable controller/Personal computer automation control system, Intech , July, pp 37-42.

BABB M. 1986 , Using a personal computer for ladder logic and documentation", Control engineering, July, pp 47-49.

BALL K. and MACZKA W.J. 1987 , "PC/Control expo report-evolution of the programmable plant, InTech, July,pp 16-20.

BAUR P.S. 1987, "PC's In control: An update of industrial software packages", Intech , July, pp 9-15.

BERG C.B. 1983 "Computer graphic displays:windows for process control", IEEE CG & A, May/June, pp 43-56.

BLACK J. 1985 , "PC ladders versus Micro languages", C & I, November, pp 155-157.

BOOCH G. "Object-oriented Development", IEEE transactions on software engineering, Vol se12, No 2, February 1986 pp 211-221

CAINE S.H. and GORDON E.K. 1980 "PDL-A tool for software design", Reprinted in tutorial on software design, edited by P.Freeman and A. Wasserman , IEEE press, pp 380-386.

CHARLAND M. and SIELAFF B. 1986, "PLC Color graphics", I & C S, March, pp 71-75.

CHIRONIS N.P. 1987 "Microprocessor-Aided equipment proves productive and reliable", Coal age, July, pp 48-56.

COX B. and Hunt B. 1986 "Objects, Icons and software IC's", No 8, August, pp 161-176.

DeWINTER F.A. and BENKE L.M. 1987 , PLC's and SCADA systems", Intech, February,pp 31-34.

DUFF C.B. 1986 "Designing an efficient language", Byte, Vol 11, No 8, August, pp 211-224.

GOULD INC 1980 Modicon 584 Programmable controller user manual, MODICON DIVISION , September.

HARTLIEB P.V. and ZILLER T. 1986 "Memory programmable control systems underground", Colliery Guardian, September, pp 413-416.

IRELAND D.G. 1987 "Detailed functional description of PLC control system for CDM no 3 Plant, recommissioning Phase 2", Internal publication of Anglo American Corp of S.A., April.

JAMSA K.A. 1984 "Object-Oriented design vs Structured design", ACM Sigsoft, Vol 9, No 1, January 1984, pp 43-49.

KISSLING J.I. 1986 ,"Personal computers, programmable controllers and the pharmaceutical industry", Advances in Instrumentation, Vol 41, Instrument Society of America.

MEYER M. 1987 "Reusability: The case for Object-Oriented design", IEEE software, Vol 4, No 2, March, pp 49-64.

NOLETT M. ,1986 " PLC programming languages: A comparison", I & C S , March, pp 57-70.

PASCOE G A. 1986 " Elements of object-oriented programming", Byte, Vol 11, No 8, August, pp 139-144.

READMANN G. 1987 ,"Is it a controller or an industrial computer", C & I , January, pp 39-41.

REEVE A.,1987 "Pursuing automation with programmable controllers", C & I, August, pp 46-47.

ROBERTS R.H. 1983 "Programmable Logic Controllers", Anglo American Corp of S.A. Internal publication , November.

SIEMENS 1986 Siemens 150K programming manual ,SIEMENS GERMANY,1986

SQUARE D 1981 Class 8884 CRT programming manual , SQUARE D ,May.

TANDY P.H. 1987 "Tripper car control philosophy", Internal publication Conlog Pty Ltd, August.

TELEMECHANIQUE 1987 TSX T607 Programming Terminal Users manual Book 4 - Grafcet language, TELEMECANIQUE FRANCE , February.

TINHAM B. 1986 , "Controller advances present and future", C & I, August , pp 39 - 44.

WAINE D. and McARTHUR N, 1987 "Documenting programs enhances PLC control", C & I, June, pp 63-64.

WALKER A.J. 1984 "Structured Digital system design", Internal Publication Dept of Elec Eng, University of the Witwatersrand, March .

WALKER A.J. 1986, "Structured information processing system design", Internal publication Dept of Elec Eng, University of the Witwatersrand, June.

WATSON K.F., O'KELLY S. and STUKKE W.F. 1987 "DE Beers Consolidated Mines - Finsch Mine - Underground control system", Internal Publication Anglo American Corp of S.A., February.

WHITE E and MALLOY R, 1986 "Object-Oriented programming", Editorial Byte, Vol 11, No 8, August, pp 137

ZARIC M. 1984 , " Structured design methodology for programmable controller applications", Internal paper University Of The Witwatersrand May 1984.

1987 Programmable controller survey, C&I, January.

APPENDIX 1

```
// Menu Definitions
#include "mainmenu.hpp"
static char * drives[] = {
                "    WHAT TYPE OF DRIVE ",
                "  1.  GENERAL DRIVE",
                "  2.  DRIVE WITH SIREN",
                "  3.  DRIVE WITH TIMER",
                "  4.  QUIT ",
                "LAST"
                };

static char * mainmenu[] = {
          "         OPTIONS MENU       ",
          "1)  ADD A NEW DRIVE  ",
          "2)  REMOVE A DRIVE   ",
          "3)  SET DRIVE INTERLOCKS ",
          "4)  DELETE DRIVE INTERLOCKS ",
          "5)  MODIFY DRIVE INTERLOCKS ",
          "6)  OPERATE A DRIVE ",
          "7)  RE - LOAD ALL INFO",
          "8)  CREATE A SEQUENCE",
          "9)  DELETE A SEQUENCE",
          "10) MODIFY A SEQUENCE",
          "11) OPERATE A SEQUENCE",
          "12) SAVE ALL INFO",
          "13) SHOW ALL DRIVES",
          "14) LIST SEQUENCES",
          "15) QUIT",
          "LAST"
          };


static char * operate[] = {
          "  Operate Menu ",
          "0) DO NOTHING",
          "1) FIELD START  ",
          "2) TRIPS HEALTHY",
          "3) STOPS HEALTHY",
          "LAST"
          };
#include "menu.hpp"
int menu(char * lines[], int * nooflines)
{
  int select,i;
```

```
    i=0;
    while (strcmp(lines[i],"LAST")!=0)
       {
        printf("%s",lines[i]);
        printf("\n");
        i++;
       }

    i=i-1;
    *nooflines = i;
    printf("MAKE A SELECTION AND HIT ENTER : ");
    scanf("%d",&select);
    return(select);
}


// Defined Class Definitions
#include "defns.hpp"
/* Digital output from the program to the real world
   The state is public as it is set from the drive routine
   The address maps the class to a physical output */

class DigO
{     public:
       int state;
       char* address;
       DigO();
};

/* Digital input from the real world to the program
   The state is private as it is normally not set by the program.
   the routine setDI is used to emulate the action of a state change
   in the real world, whilst the readDI allows read access to the state.
   The address maps the class to a physical input */


class DigI
{
   int state;
   public:

       char* address;
       DigI();
       void setDI(int value);
       int readDI();
};
```

```
/* class IOrack - contains a number of input and output . This represents the
   IO rack in the real world */

class IOrack
{
    public:
        DigO start, localstart;
        DigI fieldstart,seqstart,tripshealthy, stopshealthy, running;
        IOrack();
};


class interlocknode
{
public:
    interlocknode * next;
    int * runstate;
    char name[20];
};

class interlock
{
  public:
          interlocknode  * firstnode;
          interlock();
          list(int * nooflines);
          int setstatus();
};


//* class general drive defines the most basic drive operations. //*

class GeneralDrive
{
   public:
        char namestr[20];
        char typename[30];
        GeneralDrive * nextptr;
        interlock interlock;
        addinterlocks();
        int type;
        status();
        delinterlocks();
        virtual addtolist(char name[20]);
        setrunning();
        IOrack rack;
```

```
        virtual operate( int value, int error =0);  /* calls operate1 */
        operate1(int value, int error =0);  /* basic operations */
        GeneralDrive();
};
static GeneralDrive * temp;
static GeneralDrive * first;
static GeneralDrive * drivelist;
static GeneralDrive * list;
static GeneralDrive * current;
static FILE *fopen();

class Drivenone : public GeneralDrive
{
   public:
        operate( int value, int error =0);  /* calls operate1 */
        Drivenone() ;
        addtolist(char name[20]);
};



class Drivesiren  : public GeneralDrive
{
        sirentimer(int period);
public:
        DigO siren;
        operate(int c, int d=0);
        Drivesiren();
        addtolist(char name[20]);
};



class Drivetimer  : public GeneralDrive {
public:
        opr  ite(int c, int d=0);
        Drivetimer();
        addtolist(char name[20]);
        };


class sequencenode
{
public:
    sequencenode * next;
    sequencenode * previous;
    GeneralDrive * drive;
```

```
};


class sequence
{
  public:
          char name[10];
          sequencenode  * firstnode;
          sequence * nextseq;
          sequence();
          setup();
};

static sequence * firstsequence;
static sequence * currentsequence;



// Global Procedures
#include "listseq.hpp"

int listsequence (sequencenode * firstnode)
{
    sequencenode * currentnode;
    int i;
    i=1;
    currentnode = firstnode->next;
 do
  {
    printf ("  %d", i);
    printf ("%s",".) ");
    printf ("%s",currentnode->drive->namestr);
    printf ("%s\n",currentnode->drive->typename);
    currentnode = currentnode->next;
    i++;
  } while (currentnode!=NULL);
   return(i-1);
}


#include "listseqs.hpp"
listsequences( int * nooflines)
{
  int i,dummy;
  i = 1;
```

```
 currentsequence = firstsequence->nextseq;
 while (currentsequence != NULL)
   {  printf("%d",i,"%s"," ).  ");
      printf("%s\n",currentsequence->name);
      dummy=listsequence(currentsequence->firstnode);
      currentsequence = currentsequence->nextseq;
      i++;
   }
   *nooflines = i-1;
}


#include "addseqs.hpp"


addsequence()
{
  int select;
  currentsequence = firstsequence;
  while (currentsequence->nextseq != NULL)
    { currentsequence = currentsequence->nextseq; }
 do
   {
    select = 1;
    if (select == 1)
       {
        printf(" What is the sequence name : ");
        currentsequence->nextseq = new sequence;
        scanf("%s",currentsequence->nextseq->name);
        currentsequence = currentsequence->nextseq;
        currentsequence->setup();
       }
    printf(" Do you want another sequence 1 for yes : ");
    scanf("%d",&select);
    } while (select == 1);
}


#include "delseq..hpp"

delsequence()
{
  int select,i,count;
  sequence * temp;
  sequencenode * currentnode;
  sequencenode * tempnode;
  count = 1;
```

```
  listsequences(&i);
  printf(" Select a sequence no : (0 for none) ");
  scanf("%d",&select);
   currentsequence = firstsequence;
    if ((select > 0) & (select < i))
     {
      while ((count < i) & (count <select))
        {  currentsequence = currentsequence->nextseq;
           count++;
        }
          temp = currentsequence->nextseq;
          currentsequence->nextseq = currentsequence->nextseq->nextseq;
          currentnode = temp->firstnode->next;
          tempnode = currentnode;
          while   (currentnode !=NULL)
              { free(tempnode);
                tempnode = currentnode;
                currentnode = currentnode->next;
               }
               free(currentnode);
     }
    listsequences(&i);
}


#Include "modseqs.hpp"

modsequence()
{
  int select,i,count;
  sequence * temp;
  sequencenode * currentnode;
  sequencenode * tempnode;
  count = 1;
  listsequences(&i);
  printf(" Select a sequence no : (0 for none) ");
  scanf("%d",&select);
   currentsequence = firstsequence;
    if ((select > 0) & (select < i))
     {
      while ((count < i) & (count <select))
        {  currentsequence = currentsequence->nextseq;
           count++;
        }
          temp = currentsequence->nextseq;
       do
        {
```

```
        printf(" Options : \n");
        printf(" 1) Add a drive to the sequence : \n");
        printf(" 2) Delete a drive from the sequence : \n");
        printf(" 3) Quit : \n");
        printf(" selection : ");
        scanf("%d",&select);
        switch(select)
        {
          case(1):temp->setup();
                  break;
          case(2):count=listsequence(temp->firstnode);
                  printf(" Select a drive (0 for none)  :  ");
                  scanf("%d",&i);
//printf("debug  selected drive value %d\n",i);
                    if ((i > 0)&(i<count+1))
                    {
                      count = 1;
                      tempnode=temp->firstnode->next;
                      currentnode = temp->firstnode;
                        while (count < i)
                          {
                            currentnode = tempnode;
                            tempnode = tempnode->next;
                            count++;
                          }

                        currentnode->next=tempnode->next;
                        free(tempnode);
                    }
                  break;
          default:break;
        }
      } while (select < 3);
    }
}
#include "opseqs.hpp"


operateseq()
{
  int nooflines,select,i;
  sequencenode * currentnode;
  do
    {
      listsequences(&nooflines);
      printf(" select a sequence (0 for none : ");
      scanf("%d",&select);
```

```
        if ((select > 0) & (select < nooflines + 1))
          {    i = 1;
               currentsequence = firstsequence->nextseq;
               while ((currentsequence != NULL) & (i < select))
                  {
                    currentsequence = currentsequence->nextseq;
                    i++;
                  }
               currentnode = currentsequence->firstnode->next;
                  while (currentnode != NULL)
                     {
                       currentnode->drive->rack.seqstart.setDI(1);
                       currentnode->drive->operate(0,0);
                       currentnode->drive->status();
                       currentnode = currentnode->next;
                     }
          }
    } while (select > 0);
}



#include "add.hpp"
adddrive(GeneralDrive * current)
{
char names[20];
int  select,lines;
 while (current != NULL)
    {
          list = current;
          current = current->nextptr;
    }
       current = list;
           do
               {
                lines = 1;
                printf( "%s\n","   DRIVE TYPES ");
                list = drivelist;
                while (list != NULL)
                       {
                         printf("%d%s",list->type," )  ");
                         printf("%s\n",list->typename);
                         list = list->nextptr;
                         lines ++;
                       }
                printf("%s"," Choose a drive type (0 for none) : ");
                scanf("%d",&select);
```

```
                    if ((select > 0) &(select < lines ))
                      {
                       printf("What is the drive name : ");
                       scanf("%s",names);
                         list = drivelist;
                         lines = 1;
                           while ((list != NULL) & (lines < select))
                              {
                                list = list->nextptr;
                                lines++;

                               }
                              if ((select > 0) & (list != NULL))
                              {
                              list->addtolist(names);
                              current->nextptr = temp;
                              current = temp;
                              }
                      }
                     if (lines > select)
                      {
                        printf("%s\n","PLEASE TRY AGAIN");
                      }
                   } while (select > 0);
}

#include "listdri.hpp"
int listdrives ()
{
int i;
i=1;
current=current->nextptr;
do
{
cout << i << ".) " << current->namestr << "     "<< current->typename <<
"\n";
current = current->nextptr;
i++;
} while (current!=NULL);
return(i-1);
}


#include "finddri.hpp"
GeneralDrive * finddrives (int i)
{
int j;
```

```
j=0;
do
{
current = current->nextptr;
j++;
} while ((j!=i-1)&(current->nextptr!=NULL));
return(current);
}

#include "deldri.hpp"
deletedrive()
{
  int select;
  GeneralDrive * temp;
  listdrives();
  current = first;
  printf("\n Which drive do you want to delete : ");
  scanf("%d",&select);
  current = finddrives(select);
  temp = current->nextptr;
  current->nextptr=current->nextptr->nextptr;
  delete temp;
}

#include "operate.hpp"
operating(GeneralDrive * current)
{
 int error,select,options;
        current->nextptr->status();
          do
           {
             select = menu(operate,&options);
             current->nextptr->operate(select,error);
           } while ((select > 0) & (select < options+1));

}

operatedrive()
{
 int noofdrives, select;
 current=first;
 noofdrives = listdrives();
 printf("%s"," select a drive (0 for none) : ");
 scanf("%d",&select);
   if ((select > 0) & (select < noofdrives+1))
     {
        current=first;
```

```
        current = finddrives(select);
        operating(current);
      }
}

operatesequence()
{
   int select;
   sequencenode * current node;
   currentsequence=firstsequence->nextseq;
   currentnode = currentsequence->firstnode->next;
     while (currentnode !=NULL)
       {
        currentnode->drive->status();
        currentnode = currentnode->next;
       }
}

#include "addint.hpp"
addinterlock()
{
  GeneralDrive * thisone;
  int select,no;
 do
 {
  current = first;
  no = listdrives();
  printf("\n To which drive do you want to add the interlocks ( 0 to quit): ");
  scanf("%d",&select);
printf("debug  select : %d\n",select);
  if ((select > 0) & (select < no))
      {
          current = first;
          thisone = finddrives(select)->nextptr;
          thisone->addinterlocks();
      }
      printf(" Drive name : %s\n",thisone->namestr);
      thisone->interlock.list(&no);
 } while (select > 0);
}

#include "delint.hpp"

delinterlock()
{
  GeneralDrive * thisone;
  int select,no;
```

```
do
{
 current = first;
 no = listdrives();
 printf("\n From which drive do you want to delete an interlock ( 0 to quit): ");
 scanf("%d",&select);
 if ((select > 0) & (select < no))
      {
          current = first;
          thisone = finddrives(select)->nextptr;
          thisone->delinterlocks();
      }
 thisone->interlock.list(&no);
} while (select > 0);
}


#include "modint.hpp"

modinterlock()
{
  GeneralDrive * thisone;
  int select,no;

  current = first;
  no = listdrives();
  printf("\n Which drive's interlocks do you want to modify ( 0 to quit): ");
  scanf("%d",&select);
  if ((select > 0) & (select < no))
      {
          current = first;
          thisone = finddrives(select)->nextptr;
     do
      {
         printf(" Options : \n");
         printf(" 1) Add a drive to the interlock : \n");
         printf(" 2) Delete a drive from the interlock : \n");
         printf(" 3) Quit : \n");
         printf(" selection : ");
         scanf("%d",&select);
         switch(select)
         {
           case(1):thisone->addinterlocks();
                   break;
           case(2):thisone->delinterlocks();
                   break;
```

```
            default:break;
          }
        } while (select < 3);
      }
}

// Class procedures
#include "digital.hpp"

/* Initialisatio routine - sets the state to be 0 */

DigO::DigO()
{
     state = 0;
}

/* Initialization routine sets the state to 0 */

DigI::DigI()
{
   state = 0;

}

/*  This setDI is used to change the state of a digital input. This is
    not normally used as these state changes represent changes in the
    real world */

void DigI::setDI( int value)
{
  state = value;
}
/* ReadDI is used within the logic to ascertain the state of an input */

DigI::readDI()
{
return(state);
}


IOrack::IOrack()
{  /* upon initialization nothing is done as all IO is initialized */
     tripshealthy.setDI(1);
     stopshealthy.setDI(1);
}
```

```
#include "gendrive.hpp"
/* initialization calls the other defined classes and allows them
   to initialize themselves */

GeneralDrive::GeneralDrive()
{
        nextptr = NULL;
}

/* Operate1 forms the heart of the operation. It specifies the most
   basic drive operations  */

GeneralDrive::operate(int value, int error)
{
}

GeneralDrive::addtolist(char name[20])
{
}

GeneralDrive::operate1(int value, int error)
{  timet t0, t1;
   int temp;
   switch (value)  {
      case(0):break;
      case(1):if (rack.fieldstart.readDI()==0)
                 { temp=1;}
               else {temp = 0;}
               rack.fieldstart.setDI(temp);
               break;
      case(2):if (rack.tripshealthy.readDI()==0){
                  temp=1;   }
                  else {temp=0;}
                  rack.tripshealthy.setDI(temp);
                  break;
      case(3):if (rack.stopshealthy.readDI()==0){
                  temp=1;}
                  else {temp= 0;}
                  rack.stopshealthy.setDI(temp);
                  break;
      default:error =1;
                   }
  if (rack.seqstart.readDI()==1) {rack.fieldstart.setDI(1);}
```

```
    r a c k . l o c a l s t a r t . s t a t e      =
((((rack.fieldstart.readDI()==1)&&(rack.running.readDI()==0))||(rack.localstart.s
tate==1))&&
                        (interlock.setstatus()==1)&&

((rack.tripshealthy.readDI()==1)&&(rack.stopshealthy.readDI()==1)));

  if (rack.seqstart.readDI()==1) {rack.fieldstart.setDI(0);}

}

GeneralDrive::setrunning()
{
   int temp;
                                                t          e          m          p
=((rack.tripshealthy.readDI())&&(rack.stopshealthy.readDI())&&(rack.start.state));
    rack.running.setDI(temp);

}


GeneralDrive::status()

{
       printf("%s\n",namestr);
       printf(" field start %d",rack.fieldstart.readDI());
       printf(" trips       %d",rack.tripshealthy.readDI());
       printf(" stops        %d\n",rack.stopshealthy.readDI());
       printf(" seq start   %d",rack.seqstart.readDI());
       printf(" running      %d",rack.running.readDI());
       printf(" start out   %d\n",rack.start.state);
}


GeneralDrive::addinterlocks()
{
  GeneralDrive * tmpdrive;
  int select,no;
  interlocknode * currentnode;
  interlocknode * temp;

        do
         {
          current = first;
          no = listdrives();
          printf("\n Which drive do you want to add to the interlock ( 0 to quit):
");
```

```
        scanf("%d",&select);
          if ((select > 0) & (select < no))
              {
                currentnode = interlock.firstnode;
                 while ( currentnode != NULL)
                      {
                        temp = currentnode;
                        currentnode = currentnode->next;
                      }
                    currentnode = temp;
                    current = first;
                    current = finddrives(select);
                  interlocknode * nextnode = new interlocknode;
                  currentnode->next = nextnode;
                  (nextnode->runstate) =&(current->nextptr->rack.start.state);
                  strcpy(nextnode->name,current->nextptr->namestr);
                 nextnode->next=NULL;
                 currentnode = nextnode;
                 }
       } while (select > 0);
}


GeneralDrive::delinterlocks()
{
  int select,no;
  interlocknode * currentnode;
  interlocknode * temp;
  interlocknode * temp2;

       do
        {
         current = first;
         interlock.list(&no);
         printf("\n Which drive do you want to delete from the interlock ( 0 to
quit): ");
         scanf("%d",&select);
           if ((select > 0) & (select < no))
              {
                current = first;
                current = finddrives(select);
                currentnode = interlock.firstnode;
                                  while (( currentnode != 
NULL)&(strcmp(current->nextptr->namestr,currentnode->name)!=0))
                      {
                        temp = currentnode;
                        currentnode = currentnode->next;
```

```
                    }
                      temp2 = temp;
                      temp->next = currentnode->next;
                }
            } while (select > 0);

}


#include "drnone.hpp"

Drivenone::Drivenone()
{
     strncpy(typename,"Standard Drive",30);
     type = 1;
}

Drivenone::operate(int value, int error)
{
  int temp;
     operate1(value,error);
     rack.start.state=rack.localstart.state;
     setrunning();
     if (value > 0)
      {
      status();
      }
}

Drivenone::addtolist(char name[20])
{
   temp = new Drivenone;
   strncpy(temp->namestr,name,20);
   temp->nextptr = NULL;
}


#include "drsiren.hpp"
Drivesiren::Drivesiren()
{
        strncpy(typename,"Drive with start siren ",30);
        type = 2;
}
Drivesiren::operate(int c, int b){
       operate1(c,0);
      if ((rack.localstart.state==1)&&(rack.running.readDI()==0))
```

```
      {  siren.state=1;
         sirentimer(3);
         siren.state=0;

      }
         rack.start.state=rack.localstart.state;
         setrunning();
  if (c > 0)
     {
     status();
     }
}

Drivesiren::addtolist(char name[20])
{
   temp = new Drivesiren;
   strncpy(temp->namestr,name,20);
   temp->nextptr = NULL;
}

Drivesiren::sirentimer(int period)
{
  timet t0, t1;
  printf(" starting siren ");
  time (&t0);
  do
  {
    time (&t1);
    soundtone(5000,100,100);
  } while (t1 - t0 < period && t1 >= t0);
  printf(" siren finished ");
}


#include "drtimer.hpp"
Drivetimer::Drivetimer()
{
        strncpy(typename,"Drive with start timer",30);
        type = 3;
}

Drivetimer::operate(int c, int b){
      operate1(c,0);
      int stat,statuss,temp;
      timet t0, t1;
      if ((rack.localstart.state==1)&&(rack.running.readDI()==0))
   {
```

```
        printf (" \nHit a key to start the drive or else timeout " );
        time (&t0);
         do
        {
          time (&t1);
          statuss = (kbhit()!=0);
          if (statuss)
          {
          temp = getch();
          }
          stat = statuss||(t1 - t0 > 3);
        } while (!(stat));
        if (t1 - t0 > 3)
         rack.localstart.state=0;

    }
          rack.start.state=rack.localstart.state;
          setrunning();
  if (c > 0)
     {
     status();
     }
}
Drivetimer::addtolist(char name[20])
{
   temp = new Drivetimer;
   strncpy(temp->namestr,name,20);
   temp->nextptr = NULL;
}


#include "seqs.hpp"
sequence::sequence()
 {
  firstnode = new sequencenode ;
  firstnode->next=NULL;
  firstnode->previous=NULL;
  firstnode->drive=first;
  nextseq = NULL;
 }

sequence::setup()
{
  sequencenode * currentnode;
  int select;
   currentnode = firstnode;
   while ( currentnode->next != NULL)
```

```
    { currentnode = currentnode->next;  }
do
{
 current = first;
 listdrives();
 printf("\n Which drive do you want to add to the sequence ( 0 to quit): ");
 scanf("%d",&select);
 if (select > 0)
   {
      current = first;
      current = finddrives(select);
      sequencenode * nextnode = new sequencenode;
      currentnode->next = nextnode;
      nextnode->drive = current->nextptr;
      nextnode->next=NULL;
      nextnode->previous = currentnode;
      currentnode = nextnode;
   }
} while (select > 0);
 currentnode = firstnode;
 listsequence(currentnode);
}

#include "interl.hpp"
interlock::setstatus()
{
   interlocknode * currentnode;
   int i;
   i = 1;
   currentnode = firstnode->next;
      while (currentnode != NULL)
        {
          i =((i) & (* currentnode->runstate));
          printf(" interlock state %d",i);
          printf(" after drive  %s\n",currentnode->name);
          printf("\n");
          currentnode = currentnode->next;
        }
    return(i);
}

interlock::list(int * nooflines)
{
   interlocknode * currentnode;
   int i;
   i=1;
   currentnode = firstnode->next ;
```

```
do
 {
   printf ("  %d", i);
   printf ("%s",".) ");
   printf ("    %s\n",currentnode->name);
   currentnode = currentnode->next;
   i++;
  } while (currentnode!=NULL);
*nooflines = i-1;
}


Interlock::interlock()
 {
//  firstnode = NULL;
  firstnode = new interlocknode ;
  firstnode->next=NULL;
  strcpy(firstnode->name,"first");
  *firstnode->runstate = 1;
 }



// File Handling
#include "drivlist.hpp"
drivelist()
{
  Drivenone * gen = new Drivenone;
  strncpy(gen->namestr,"gen",20);
  gen->nextptr= NULL;
  Drivesiren * siren = new Drivesiren;
  strncpy(siren->namestr,"siren",20);
  siren->nextptr= NULL;
  Drivetimer * timer = new Drivetimer;
  strncpy(timer->namestr,"timer",20);
  timer->nextptr= NULL;
  drivelist = gen;
  gen->nextptr= siren;
  siren->nextptr= timer;
}


#include "fileadd.hpp"
fadd()
{

char names[20];
```

```
int  select,lines,options;
FILE *fin;
char fname[11];
 printf(" Do you want to change the file name (1 for yes) : ");
 scanf("%d",&select);
 if (select != 1)
   {
    fin = fopen("data.txt","r");
   }
  else
    { printf (" What is the file name : ");
      scanf("%s",fname);
      fin = fopen(fname,"r");
    }
 if (!fin)
  {printf(" error in file ");}

// reads drives
printf(" Reading drives \n");
    current = first;
          list = drivelist;
           options = 0;
            while (list != NULL)
              {
                list = list->nextptr;
                options ++;
              }
   fscanf(fin,"%d",&select);
    if (select > 0)
    {
       do
         {
                fscanf(fin,"%s",names);
printf(" Reading : %s\n",names);
                list = drivelist;
                while ((list != NULL) & (list->type != select))
                   {
                      list = list->nextptr;
                   }
                if (list != NULL)
                   {
                      list->addtolist(names);
                      current->nextptr = temp;
                      current = temp;
                   }
            fscanf(fin,"%d",&select);
         } while (select > 0);
```

```
    }

// reads sequences
printf(" Reading sequences \n");
sequencenode * currentnode;
char seqname[10];
char driname[20];
    currentsequence = firstsequence;
    fscanf(fin,"%s",seqname);
   printf(" Reading : %s\n",seqname);
    if (stromp(seqname,"zz")==0)
     {
       fscanf(fin,"%s",seqname);
   printf(" Reading : %s\n",seqname);
       while (stromp(seqname,"zzzz")!=0)
          {
             currentsequence->nextseq = new sequence;
             strcpy(currentsequence->nextseq->name,seqname);
              fscanf(fin,"%s",driname);
   printf(" Reading : %s\n",driname);
              currentnode = currentsequence->nextseq->firstnode;
             while (stromp(driname,"zz")!=0)
             {
               current = first;
               while (stromp(driname,current->nextptr->namestr)!=0)
                 {
                 current = current->nextptr;
                 }
                   sequencenode * nextnode = new sequencenode;
                   currentnode->next = nextnode;
                   nextnode->drive = current->nextptr;
                   nextnode->next=NULL;
                   nextnode->previous = currentnode;
                   currentnode = nextnode;
                   fscanf(fin,"%s",driname);
   printf(" Reading : %s\n",driname);
             }
              fscanf(fin,"%s",seqname);
   printf(" Reading : %s\n",seqname);
             currentsequence= currentsequence->nextseq;
          }

     }


char drname[20];
```

```
GeneralDrive * tmpdrive;
interlocknode * currentnod;
printf(" Reading interlocks \n");

   fscanf(fin,"%s",drname);
   printf(" Reading : %s\n",drnam.
   if (stromp(drname,"xx")==0)
     {
       fscanf(fin,"%s",drname);
   printf(" Reading : %s\n",drname);
       while (stromp(drname,"xxxx")!=0)
          {
           current = first->nextptr;
              while(stromp(drname,current->namestr)!=0)
                {
                  current = current->nextptr;
                }
           tmpdrive = current;
           currentnod = tmpdrive->interlock.firstnode;
            do
              {
                fscanf(fin,"%s",drname);
   printf(" Reading : %s\n",drname);
                current = first->nextptr;
                if (stromp(drname,"xx")!=0)
                {
                    while(stromp(drname,current->namestr)!=0)
                      {
                        current = current->nextptr;
                      }
                  interlocknode * nextnod = new interlocknode;
                  currentnod->next = nextnod;
                  (nextnod->runstate) =&(current->rack.start.state);
                  strcpy(nextnod->name,current->namestr);
                  nextnod->next=NULL;
                  currentnod = nextnod;
                }
              } while (stromp(drname,"xx")!=0);
            fscanf(fin,"%s",drname);
   printf(" Reading : %s\n",drname);
          }
     }
fclose(fin);
}
```

```
#include "filesave.hpp"
fsave()
{
FILE *fout;
int select;
char fname[11];
 printf(" Do you want to change the file name (1 for yes) : ");
 scanf("%d",&select);
 if (select != 1)
   { fout = fopen("data.txt","w");  }
  else
    { printf (" What is the file name : ");
      scanf("%s",fname);
      fout = fopen(fname,"w");
    }
 if (!fout)
  {printf(" error in file ");}
// saves all drives


current=first->nextptr;
while (current!=NULL)
   {
    fprintf(fout,"%d\n",current->type);
    fprintf(fout,"%s\n",current->namestr);
    printf(" saving %s\n",current->namestr);
    current =current->nextptr;
   }
        fprintf(fout,"%d\n",0);
        fprintf(fout,"%s\n","zz");


// saves all sequences
 sequencenode * currentnode;
 currentsequence = firstsequence->nextseq;
  while (currentsequence != NULL)
    {  printf(" saving ");
      printf("%s\n",currentsequence->name);
      fprintf(fout,"%s\n",currentsequence->name);
      currentnode = currentsequence->firstnode->next;
        do
        {
          fprintf(fout,"%s\n",currentnode->drive->namestr);
          currentnode = currentnode->next;
        } while (currentnode!=NULL);
        fprintf(fout,"%s\n","zz");
        currentsequence = currentsequence->nextseq;
```

```
  }
 fprintf(fout,"%s\n","zzzz");



 current=first;
 interlocknode * currentnod;
  while (current!=NULL)
    {
      if (current->interlock.firstnode->next!=NULL)
          {
            currentnod = current->interlock.firstnode->next;
              fprintf(fout,"%s\n","xx");
              fprintf(fout,"%s\n",current->namestr);
              while (currentnod != NULL)
                 {
                  printf(" saving interlock: %s\n",currentnod->name);
                  fprintf(fout,"%s\n",currentnod->name);
                  currentnod = currentnod->next;
                 }
          }
       current = current->nextptr;
    }
 fprintf(fout,"%s\n","xx");
 fprintf(fout,"%s\n","xxxx");
 fclose(fout);
}


// Main menu procedure
#include "firstmen.hpp"
firstmenu()
{
 int select=0;
 int nooflines=0;
 int dummy,i;
 do
  {
   select = menu(mainmenu, &nooflines);
  printf(" debug select %d\n",select);
  current = first;
    currentsequence = firstsequence;
    switch(select)
     {
    case(1):adddrive(current);
            break;
```

```
    case(2):deletedrive();
            break;
    case(3):addinterlock();
            break;
    case(4):delinterlock();
            break;
    case(5):modinterlock();
            break;
    case(6):operatedrive();
            break;
    case(7):fadd();
            break;
    case(8):addsequence();
            break;
    case(9):delsequence();
            break;
    case(10):modsequence();
            break;
    case(11):operateseq();
            break;
    case(12):fsave();
             break;
    case(13):listdrives();
              printf("strike any key when ready");
              dummy = getch();
              break;
    case(14):listsequences(&i);
            printf("strike any key when ready");
            dummy = getch();
            break;
     default:break;
     }
   } while (select < nooflines);'
}

// Main Body

main()
{

  drivelist();

  first = new GeneralDrive;
  strncpy(first->namestr,"first",20,;
  strncpy(first->typename,"first",10);
  first->nextptr=NULL;
  current = first;
```

```
  firstsequence = new sequence;
  strncpy(firstsequence->name,"firstseq",10);
  firstsequence->nextseq = NULL;

// Actual Main Body

  fadd();
  firstmenu();
  fsave();


} /* main end */
```

Author: Bricker, Rael I.
Name of thesis: Object oriented development, a logical approach to control system software design.

PUBLISHER:
University of the Witwatersrand, Johannesburg
©2015

LEGALNOTICES:

Copyright Notice: All materials on the University of the Witwatersrand, Johannesburg Library website are protected by South African copyright law and may not be distributed, transmitted, displayed or otherwise published in any format, without the prior written permission of the copyright owner.

Disclaimer and Terms of Use: Provided that you maintain all copyright and other notices contained therein, you may download material (one machine readable copy and one print copy per page)for your personal and/or educational non-commercial use only.

The University of the Witwatersrand, Johannesburg, is not responsible for any errors or omissions and excludes any and all liability for any errors in or omissions from the information on the Library website.