AN OBJECT ORIENTED APPROACH TO LIBRARY INFORMATION SYSTEM (O O L I S)

A PROJECT REPORT SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE AWARD OF THE DEGREE OF

MASTER OF TECHNOLOGY IN COMPUTER SCIENCE

BY

K. SESHAGIRI **R. BALAJI**

DEPARTMENT OF COMPUTER & INFORMATION SCIENCES

SCHOOL OF MATHEMATICS & COMPUTER / INFORMATION SCIENCES

UNIVERSITY OF HYDERABAD

HYDERABAD - 500 134, INDIA

JANUARY 1995

CERTIFICATE

This is to certify that the project titled "An Object Oriented Approach to Library Information System (OOLIS)" submitted to the school of Mathematics and Computer / Information sciences, in partial fulfillment for the award of **MASTER OF TECHNOLOGY** degree in **COMPUTER SCIENCE,** is the bonafide work of **R.BALAJI** and **K.SESHAGIRI** under my guidance.

The matter embodied in this report is the original work of above said and has not been submitted to any other University / Institution for the award of any other degree / diploma.

[illegible]

Dr. P.R.K. Murthy,
Internal Guide
Reader, Dept. of Computer/
Information sciences,
University of Hyderabad.

[illegible]

Prof. A.K.Pujari,
Head
Dept.of Computer/
Information sciences,
University of Hyderabad.

[illegible]

Prof. V.Kannan,
Dean, School of MCIS,
University of Hyderabad,
Hyderabad-500 134.

ACKNOWLEDGEMENT

Our sincere thanks go to **Dr. B.R.K.Murthy, Reader, H.C.U**, who poured over every inch of the project, and manuscript with painstaking attention to detail and made a semi-infinite number of helpful suggestions.

Next our thanks go to **Dr. A. S. Reddy, Reader, H.C.U**, who reviewed the design, its suitability for implementation and suggested many excellent improvements.

We acknowledge with reverence the help extended to us by **Dr. V.Kannan**, Dean , School of MCIS and **Prof. A.K.Pujari**, Head, Dept.of CIS,. We would also at this point of time like to thank the members of Indira Gandhi Memorial Library and A.I.Lab, H.C.U. for providing invaluable help without which this project wouldn't have reached fruition.

Finally, we would place our gratitude to **Ms. ArunPriya** for ably shepherding the manuscript through production, **Mr.D.S.Somayajulu**, **Mr.V.Sreeram**, **Ms. N.Sudha**, **Ms.K.V.Pratima**,**Mr.R.Narayanan**, and **OUR PARENTS** who stood by us when we were crossing troubled waters.

R. Balaji
K. Seshagiri

FOREWORD

As computer professionals, we strive to build systems that are useful and that work, as software engineers, we are faced with the task of creating complex systems in the presence of scarce computing and human resources. Over the past several years, OO technology has evolved in diverse segments of the computer science as means of managing the complexity inherent in many different kinds of systems. The object model has proven to be a very powerful and unifying concept.

Object Oriented Programming (OOP) is the most dramatic innovation in software development in the last decade. It ranks in importance with the development of the first higher-level languages at the dawn of the computer age. Sooner or later every programmer will be affected by the object-oriented approach to program design.

It is under this [illegible] an object oriented approach to the library information system has been made. To

1) obtain an higher level of abstraction that appeal s to the workings of human cognition,
2) make the reusability of not only the software bu t also the entire design,
3) make the system be resilient to changes,

an object oriented approach to the library i nformation system has been ventured upon.

ABSTRACT

Object Oriented Design is the current trend in the design of complex software systems. Many object oriented methodologies have emerged, based on the same fundamental concepts and differing only in stress on certain principles and heuristics during the development of the system.

In this project the Library Information System is being captured using an object model. The various components in this organisation (Library) are thought of as objects, and an Object Oriented design is evolved. For example, Book can be thought of as an object with its own attributes like Number, Title, Author, Price etc.,and a Class Template Books with Member Data and Functions can be defined. Further Borrowable Books, Reference Books, Periodicals, Backvolumes are all basically Books, but have some extra features for themselves. To quote a few, Borrowable Books can be borrowed for stipulated time period, have fine for late returns or renewals, Reference Books cannot be taken out of the library, have some fixed time period for referring them, Periodicals have periodicity, subscription information associated with them , Backvolumes have a set of periodicals bound together and so on. So Books class becomes a Base class and all other types of Books(Borrowable/ Reference/ Periodicals/ Backvolumes) are derived from this Base class and add their own extensions. In this manner the various objects in a typical library are identified and suitable class definitions given. Then the relationship amongst these classes, how they interact with each other by passing messages are established.

A Class diagram capturing all essential details is evolved, and based upon this design implementation is done.

The various features of Object Oriented Programming(OOP), like Data Hiding and Encapsulation, Inheritance, Containership, Reusability, Polymorphism, Operator and Function overloading are all being exploited in the design and implementation of the Object Oriented Library Information System (OOLIS).

CONTENTS

Chapter 1

INTRODUCTION

1.1 Why design methodologies ?

What is design ?

In simple terms, *design* is the disciplined approach we use to invent a solution for some problem, thus providing a path from requirements to implementation.

As Stroustrup suggests, "the purpose of *design* is to create a clean and relatively simple internal structure, some times also called an architecture" [3].

Design involves balancing a set of competing requirements. The products of design are models that enable us to reason about our structure, make trade-offs when requirements conflict and in general, provide a blueprint for implementation.

What is methodology ?

A method is a disciplined process for generation of a set of models that describe various aspects of a software system under development, using some well defined notation. A *methodology* is a collection of methods applied across the software development life cycle and unified by some general approach.

If a project is small, or non-critical, or say both, a methodology may not be important. But software is being used increasingly in applications(generally very big) where reliability and safety are essential.

We can *systematically* and ***predictably*** arrive at solutions that are pragmatic, cost-effective, and timely to real world problems only if we follow certain set of well defined and tested techniques.

1.2 Why Object Oriented design methodologies ?

"**Catastrophy is a friend of mine** " - Any complex system.

Complexity:

The challenge of developing **industrial strength** software, applications that exhibit a very rich set of behaviors as, for example, system that have to maintain integrity of hundreds of thousands of records of information while allowing concurrent updates and queries for command and control of real world entities, such as the air or rail traffic.

The unique and distinguishing characteristics of **industrial strength** software is that it is intensely difficult, if not impossible for the individual developers to comprehend all the subtleties of the design. Plainly, Complexity is an essential property of all large software systems. Complexity of such systems exceed the human intellectual capacity.

As we first begin to analyze a complex software system, we find many parts that must interact in a multitude of intricate ways with little perceptible commonality among either part or their interactions we work. To bring organization to the complexity through the process of design, we must think about many things at once, thus complexity of software system we are asked to develop is increased, yet there are basic limits upon our ability to cope with this complexity.

How to manage the complexity ?

It is natural for us to look how the problems we face, are solved by others or by nature itself.

"Divide et impera " - classic principle.

As the above principle advocates if you can't face problems in the complete form, divide and conquer it.

We view the world as a set of autonomous agents(or objects) that collaborate to perform some higher level of behavior. Each object in solution embodies its own unique behavior and each one models some object in the real world. From this perspective, an object is simply a tangible entity which exhibits some well defined behavior.

t Oriented design:

nalogy between objects in object oriented design and
oblem domain result in systems that are easier to
makes the design more intuitive and simplifies
: requirements and software code,

d approach results in software that is easily extended,
aintained.
d decomposition yields smaller systems through the
n mechanisms thus providing an important economy

d systems are more resilient to changes and thus are
over a period of time because their design is based
ermediate forms.

Benefits of Objec

The direct a
the objects in prc
understand. This
traceablity betweer

- Object Oriente
  modified and m
- Object Oriente
  use of commo
  of expression.
- Object Oriente
  able to evolve
  upon stable inte

- Object Oriented decomposition greatly reduces the risk of building complete software system because they are designed to evolve increasingly from smaller system in which already have confidence.

1.3 Overview of Booch methodology of OOD

Booch methodology

Process Design:

Clean internal structure is essential to construct a system that is understandable, can be extended, reorganized, maintainable and testable.

An iterative and incremental development life cycle is the antithesis of traditional waterfall life cycle, and so represent neither a strictly top down nor a bottom up process.

Object Oriented development is neither strictly top down nor strictly bottom up, but is round trip gestalt design which emphasizes the incremental iterative development of the system through the retirement of different logical and physical views of the system as a whole. Round-trip gestalt is the foundation of the process of Object Oriented design.

Booch is reluctant to prescribe a fixed ordering of phases for object oriented design. Rather he recommends that the analyst work iteratively, and incrementally augment formal diagram with informal techniques as appropriate to the problem at hand. Nevertheless, Booch delineates four steps that must be performed during the course of object oriented design.

In this context, Rumbaugh remarks that " when you think that the design is complete at one level of abstraction, add more details and flesh at the design further at a finer level of details. You may find that new operations and attributes must be added to classes and possibly new classes will be identified".[14]

The development process:

The different phases of the software project, such as design, implementation and testing can be strictly separated.

1. Identify the classes and objects at a given level of abstraction.
2. Identify the semantics of these classes and objects.
3. Identify the relationship among these classes and objects.
4. Specify the interface and then the implementation of the classes and objects.

Identify the classes and objects at a given level of abstraction:

Purpose:

The purpose of identifying classes and objects is to establish the boundaries of the problem . This activity is the first step in devising an object oriented decomposition of the system under the development.

As a part of the analysis we apply this step to discover those abstraction that form the vocabulary of the problem domain and by so doing we begin to constrain our problem by what is what is not of interest. As part of design we apply this step to invent new abstractions that form elements of lower level abstraction that we can use to construct high level ones and to discover commonality among existing abstractions, which we can exploit in order to simplify the system's architecture.

Some of the classes and objects we identify early in the life cycle will be wrong , but that is not necessarily a bad thing. As we learn more about the problem, we will probably change the boundaries of certain abstractions by allocating responsibilities, combining similar abstractions, and dividing larger abstractions into groups of collaborating ones.

Identifying the semantics of classes and objects:

Purpose:

The purpose of identifying the semantics of classes and objects is to establish the behavior and attributes of each abstraction identified in the previous phase. Here we refine our candidate abstractions through an intelligent and measurable distribution of responsibilities.

As part of analysis we apply this step to allocate the responsibilities for different system behaviors. As part of design, we apply this step to achieve a clear separation of concerns among the part of our solution. As implementation proceeds, we move from free form descriptions of roles and responsibilities to specifying a concrete protocol for each abstraction.

Early in the development process, we may specify the semantics of classes and objects by writing the responsibilities for each abstraction in free form text. As we refine the protocol of each abstraction, we can introduce finite state machine for certain classes, especially for those that involve event driven state ordered behavior so as to capture the dynamic semantics of the protocols.

Identifying the relationships among classes and objects:

Purpose:

The purpose of identifying the relationships among classes and objects is to solidify the boundaries , and to recognize the collaborations with each other abstractions identified earlier.

As part of analysis we apply this step to specify the association among classes and objects. As part of design, we apply this step to specify the collaborations that form the mechanisms of our architecture, as well as the higher level clustering of classes into categories and modules into subsystems. As implementation proceeds, we refine relationships such as association into more implementation oriented relationship like instantiation and use.

Implementing classes and objects:

Purpose:

During analysis, the purpose of implementing classes and objects is to provide a refinement of existing abstraction, sufficient to umviesl classes and objects at the next level of abstractions. During design the purpose of this activity is to create tangible representations of our abstractions.

Primary activity associated with this step is the selection of the structure and algorithm that provide the semantics of the abstraction we identified earlier. We may capture our analysis and design decisions regarding these classes and objects and their collaborating according to two dimensions: their *logical/physical* view, and their *static / dynamic* view.

Both dimensions are necessary to specify the structure and behavior of on object oriented system.

Logical view of a system describe the existence and meaning of the key abstraction and mechanism that form the problem space or that define the system's architecture. The *physical* model of a system describes the concrete software and hardware composition of the system's context of implementation.

Events happen dynamically in all software intensive system. Objects are created and destroyed, objects send messages to one another in an orderly fashion and in some systems, external trigger operations upon certain objects. In object oriented development, we express the dynamic semantics of a problem or its implementation in this dynamic view of the system.

1.4 Problem Definition

The aim of this project is to capture the Library Information System using an Object model. The various objects present in LIS are identified and the relationships amongst them established using Booch methodology of Object Oriented Design.

The end result is to develop a software system satisfying and solving the operations of a typical Library.

End users:

Library personnel and the users of library service.

Environment:

Implementation is done using Borland C++ , version 3.1.

Assumption:

Reader is familiar with Object-Oriented Methodology. For an exhaustive treatment of Object Oriented concepts, please refer to Bibliography.

Scope:

Vast scope is present to extend the project. As the Design is given it can be refined, redefined etc., Windows interface can be given for making the software user friendly.

1.5 Organization of the report

Organization of the report is as follows,

Chapter 1 Introduction : In this chapter, need for design methodologies, reason for object oriented approach, overview of Booch methodology, its advantages are all presented. Finally the problem is defined.

Chapter 2 System study : This chapter goes into the details of a typical library information system and the essential features that are to be captured in the design.

Chapter 3 Design : This chapter starts with an overview of the construction of the problem domain classes. Then the design itself is presented as a class diagram, after which a descriptive explanation of the design follows.

Chapter 4 Implementation : This chapter gives the essential details about the concepts involved in the Implementation and the implementation itself.

Chapter 5 Conclusions : This chapter gives highlights about the observations, limitations and future scope of the system.

At the end **Bibliography** is given as Appendix.

Chapter 2

SYSTEM STUDY

2.1 System

Definition:

In simple words "System" is defined as, **"collection of inter-related elements which work together to achieve a common goal".**[21]

Characteristics:

A System mainly consists of three components. They are the input,output and the process. The System is supplied with data and instructions known as the input. Then it processes the information to produce the output. Hence the system may also be considered as a set of elements responding to produce output.

[illegible] is not an element of the system belongs to the environment. ...

2.2 Information System

An information system is a set of organized procedures that, when executed, provide the necessary information for decision making and/or control of the organization. Information System is one that processes input, maintains files of data and produces information, reports and other output.

Information system consist of subsystems, including hardware, software and data storage for files and databases. The particular set of subsystems used - the specific equipments, programs, files and procedures - constitutes an information systems application.

Types of information systems :

There are several different types of information systems to meet a variety of needs. Some of them are

Transaction Processing System:

The most fundamental computer based information system pertains to the processing of business transactions. Transaction processing systems (TPS) are aimed at improving the routine business activities on which all organizations depend. A transaction is any event or activity that affects the organization.

Management Information System:

In contrast, T[illegible] [illegible]

Management Information Systems (MIS) assist managers in decision making and problem solving. They draw on data stored as a result of transaction processing, but they may also use other information.

Decision Support system:

Decision Support Systems (DSS) assist managers who must make decisions that are not highly structured, often called unstructured or semi-structured decisions. A decision is considered unstructured if there are no clear procedures for making the decision and if not all the factors to be considered in the decision can be readily identified in advance. A key factor in the use of Decision support system is determining what information is needed.

2.3 A trip into the world of libraries

Library is a social institution charged with the most enviable function of dispensing knowledge to the ignorant and the informed alike. "*vidya_dan*" ,i.e. imparting of knowledge has been considered as the most sacred task in ancient India and Manu rightly allotted this job to [illegible] indispensable had a pride of place in society. The administration of such an important social institution naturally demands greater attention than given in the past.

The main function of a library is the collection and preservation of knowledge for its dissemination to all. Its conservation for posterity is also an important duty of a library.

Departments of library:

The entire library can be broadly fit into three important departments[17], each of which can further be divided into sections. They are

Technical department

Acquisition section

Processing section

Classification

Cataloguing

Periodicals section

Service department

Circulation section

Reference section

Miscellaneous department

Personnel Administration section

Finance & Budgeting section

2.3.1 Technical department

This section, generally does not deal with the reading public directly. Its work is properly described as " behind the scene work". The staff employed here are engaged in processing the books and other kindred materials by using accepted techniques such as accessioning, classifying and cataloguing so that these are used by the public.

Acquisition Section:

Knowledge is power and libraries are the reservoirs of this power. This power is contained in Books in the form of kinetic power. "Acquisition of Books is a prerequisite for a library".

Acquisition is so important that it should be organized in such a manner that the reading material of maximum utility is acquired without any delay and at a minimum cost. This can be ensured only if a suitable section is organized and the latest Acquisition techniques are used. Some of the functions of the Acquisition Section are the

1. Maintenance and use of bibliographies aids peculiar to acquisition work (e.g. Dealers Catalogues, Trade Lists etc.,).
2. Maintenance of Order Files and other records essential to acquisition work.
3. Making up, Despatching and Filing of orders for Books.
4. Receipt, Handling and Inspection of arrivals.
5. Preparation of Bills for payment, Book Keeping and other activities.
6. Informing individuals regarding the purchase of Books and its status.

7. Following up on items not promptly received.

Thus Acquisition Section assumes a pivotal position in the Organization of Libraries.

Processing Section:

Classification:

Classification is the foundation of librarianship. Classification, in common parlance means arrangement. To state it more clearly it means "to bring order out of chaos". If a permanent order of classification of books is not brought in to the library, the time spent in arranging, rearranging ,dearranging will be so much that the readers will become sore about the libraries and will begin to detest the library service. Various classification schemes have been devised each of which has its own merits and demerits. The important of them are.

- Dewey Decimal Classification by Melvil Dewey.
- Universal Decimal Classification .
- Library of Congress Classification.
- Bibliographic Classification by H.E.Bliss
- Rider's International Classification by Fremont Rider.

The Classifying process goes as below.

1. Duplicate Checking.
2. Determining Specific Subject of the books.
3. Allotting Class numbers by referring to the Classification Schedule.
4. Assigning Book Numbers
5. Assigning Subject Headings.
6. Checking of Class Numbers and Subject Headings by the

Chief Classifier.

7. Maintenance of Staff Manual.

Cataloguing:

Classification determines the place of a Book on the Shelves. Physically, Book can be placed only at one place according to Subject contents of the Book, but the readers seek the Book through various approaches such as Author, Title, Series and other Collaborators like Translators, Editors etc.,. Classification is not capable of fulfilling these varied approaches of the readers. The catalogue comes to the rescue of the readers.

Catalogue is a guide to the Blind Alley. As a City without a map is difficult to be known about, so is a library without a Catalogue. It is needless to emphasize the importance of a Catalogue because without it the whole purpose of libraries will be lost. Books will not be put to as best use as is desirable.

Cataloguing Procedure:

1. Preparation of Main entry.
2. Preparation of Shelf list.
3. Label Pasting.
4. Label Writing and assigning location marks.
5. Checking of the Catalogue cards by the Chief Cataloguer
6. Alphabetization of cards.
7. Card Filing.
8. Maintenance of Staff Manual and Authority files.

Periodicals Section:

The Periodical Section deals with the Selection, Procuring and Maintenance of Periodicals and Journals. Periodicals can be distinguished among themselves on the grounds of their literary contents or their sponsoring bodies. Keeping in view the former criterion , periodicals may be classified as under:-

1. Those intended to foster the interest of Knowledge

2. Those intended to foster the interests of a trade, profession or society.

3. Money making ventures.(Intended for popular appeal)

According to the second criterion and the British practice, the broad division of Periodicals may be as under:-

1. The Publications of Societies and other organizations.

2. House Journals.

3. The Independent Periodicals.

Magazines, Periodicals and Journals are published at various intervals. Their frequency of publication may be Daily, Bi-weekly, Weekly, Fortnightly, Monthly, Bi-Monthly, Quarterly, Half-Yearly, Annual or irregular.

Periodicals and Magazines can be acquired by,

1. Subscription
2. By becoming a member of Societies and learned Institutions.
3. By Gift.
4. By exchange.

Before procuring Periodicals and Magazines the types of users of the library are taken into account and based upon it the selection is done.

2.3.2 Service Department:

Circulation Section:

"Of all the library activities, the Circulation of Books for home use represents by far the major Public Service rendered by library"[16]. Following are the important jobs of Circulation Section of libraries.

1. Registration of Members.
2. Lending of Books.
3. Charging of Overdues.
4. Reservation of Books.
5. Renewal of Books.
6. Maintenance of Records.
7. Maintenance of Statistics.
8. Lending of Books on Inter Library Loan Basis.
9. Miscellaneous jobs.

Thus the Circulation Section can be thought of as the heart of library service.

Reference Section:

"Reference is to library service as intelligence is to military service"

Reference Section is hub of all the activities of the library. Reference Service is " A Sympathetic and informed personal aid in interpreting library collection for study and research". William Warner Bishop[24] defines reference work as " the service rendered by the librarian in aid of some sort of study. It is not the study itself-that is done by the reader, but the help given to the reader engaged in research [illegible] "

The Reference Service sho uld keep in mind the various readers [illegible] provide effective availability and accessibility of materials.

2.3.3 Miscellaneous Department

Even though the following sections are termed the Miscellaneous they also form complimentary part of the library organization.

Personnel Administration Section:

It is men who run an organization. It is they who convert materials into salable commodities. The energies of these men are to be channeled in right directions. Personnel administration implies a process of getting the best out of the employees of an organization by means of judicious selection, tactfully dealing and by selecting their replacement, if necessary. The functions of personnel management are

1. Job analysis and Evolution
2. Staffing
3. Recruitment and selection
4. Tests
5. Placement
6. Induction
7. Training

Financial Section:

Finance is the motive power. Libraries, unlike other state or central govt. and/or local govt. departments, are not revenue fetching agencies. Rather, on the otherhand, these are spending institutions, because these partake the nature of nation-building departments. Further libraries are growing organizations, implies that, the books, readers, staff, buildings and furniture grow day by day.

So to effectively manage such an organism the finance section should identify, the various incomes and expenditure and to maintain an equilibrium among them.

The main source of the public library revenues are

1. Subscriptions
2. Endowments and private benefactions.
3. Library rates.
4. Government grants.
5. Gifts.
6. Fees and Fines.

The expenditure is divided as under

1. Salaries and Wages.
2. Books.
3. Periodicals and News papers.
4. Binding.
5. Heating and Lighting etc.,
6. Rents, Loans, Insurances etc.,
7. Other Miscellaneous charges ,

The above chapter gives us the gist of a typical library and the essential details that are to be captured in the design.

Chapter 3

DESIGN

"Great designs come from great designers, not from great tools"

Overview:

To achieve the aim described in the earlier chapter, as in the case of any system, initially problem domain classes are constructed by analyzing the concepts advocated by the methodology.

3.1 Constructing Problem Domain Classes

Introduction

Problem Domain Classes represent the abstractions and relations among these abstractions using which designer describes the static aspects of his design e.g. class, class utility, object inheritance etc.,

As explained earlier, the problem domain classes for this project are nothing but *abstractions* and *relations* among these abstractions. Structure of these Classes consist of those details that are captured from the designer as suggested by the methodology in not so exact manner. Functionality of those classes include ways to get data from the user, ways to validate this information, ways to present themselves back to the user.

Here, how the structure of these classes is established is described.

3.1.1 Elements of the Object model:

Each of the styles of programming design is based upon its own conceptual framework. For all things object-oriented, the conceptual frame work is the *object model*.The elements of *Object Model* are described here.

•Abstraction, Encapsulation and Hierarchy:

•Abstraction :

One of the fundamental ways with which humans cope up with complexity is through abstractions. Hoare suggests that " Abstractions arise from a recognition of similarities between certain objects, situations or processes.in the real world, and the decision to concentrate upon these similarities and to ignore for the time being the differences".[13]

An abstraction focuses on the ***outside*** view of an object and so serves to separate an object's ***essential*** behavior from its implementation.

Some kind of abstractions observed are

Entity abstraction:

An object that represents a useful model of a problem domain or solution domain entity.

Action abstraction:

An object that provides a generalized set of operations, all of which perform the same kind of function.

We strive to build entity abstractions, because they directly parallel the vocabulary of the problem domain.

•Encapsulation:

Abstraction and encapsulation are complimentary concepts. Abstraction focuses upon the observable behavior of an object whereas encapsulation focuses upon the implementation that gives rise to this behaviour. While abstractions help people to think about what they are doing, encapsulation allows program changes to be reliably made with limited effort.

Liskov remarks that "for abstractions to work implementations must be encapsulated"[15]. In practice, this means that each class must have two parts one interface and another implementation. The interface of a class captures only its outside view, encompassing our abstractions of the behaviour common to all instances of the class, the implementation of the class comprises the representation of the abstraction as well as the mechanisms that achieve the desired behavior.

- *Hierarchy*:

We may find many abstractions in problem domain more than what we can comprehend at one time. A set of abstractions often forms a hierarchy, and by identifying these hierarchies in our design, we greatly simplify the understanding of the problem.

The above three principles results in to abstractions like classes. objects and relations among the abstractions.

3.1.2 Fundamental Principles:

3.1.2.1 *Object*:

In simpler terms, Object can be defined as "A *Tangible* entity that exhibits some well-defined behavior". But real world objects are not the only kind of objects that are of interest to us during the software development. Other important kinds of objects are inventions of the design process whose collaborations with other such objects serve as the mechanisms that provide some higher level behaviour

Objects possess *state* which reflects values of data abstracted at that particular instant and *behavior* which is the role of the object Behavior is how and Object acts and reacts. in terms of its state changes and message passing.

Life-span of an object:

The life-time of an Object extends from the time it is first created until that space is reclaimed. But certain objects may be *persistent* meaning that their lifetime transcends the life time of the program that created them.

As we identify many objects whose structure and functionality are alike, we characterize the objects with its class.

Relationship among Objects:

These relationships are typical **client/server** relationships. Objects communicate with one and other by sending/receiving messages. The messages that one object sends to another object depends upon the methods accessible to its class.

Relation is characterized by the set of messages attached to it.

Visibility of objects across a link:

Consider two objects A and B with a link between the two. Inorder for A to send message to B, B must be visible to A in some manner. Visibility of an object across a link may be one of the following four.

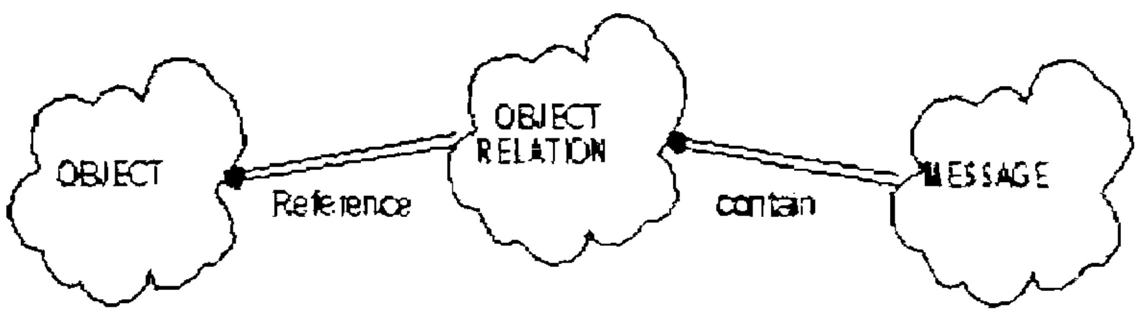
(*Supplier* object is the object whose services are utilised. *Client* Object is the object using the services.)

- The *Supplier* object is global to the *Client*.
- The *Supplier* object is a parameter to some operation of the *Client*.
- The *Supplier* object is a part of the *Client* object.
- The *Supplier* object is a locally declared object in some operation of the *Cient*.

Synchronization:

Whenever one object passes a message to another across a link, the two objects are said to be synchronized. For objects in a completely sequential application, this synchronization is usually accomplished by simple method invocation. However in the presence of multiple threads of control, objects require more sophisticated message passing in order to deal with the problem of mutual exclusion that can occur in the concurrent systems.

Synchronization details are just captured, no checks are employed.To employ checks and facilitate other editing(delete,move...) features, objects should have references to object-object relationships.



Some of the checks that may be performed in an object diagram where the designer describes messages flow among the objects are

- If an object that is created, belongs to a valid class(in the class diagram,that class should be deleted during the refinement of the process).
- When an object A sends a message to B, that corresponding message should be in B's class and is accessible to A's class.
- If the persistence is compatible with the class persistence.
- If objects are without relationships(completeness check)

3.1.2.2 Class:

Class is an Abstraction over objects,"the very essence" of an object. While an individual object is a concrete entity that performs some role in the overall system, the class captures the structure and behavior common to all related objects.

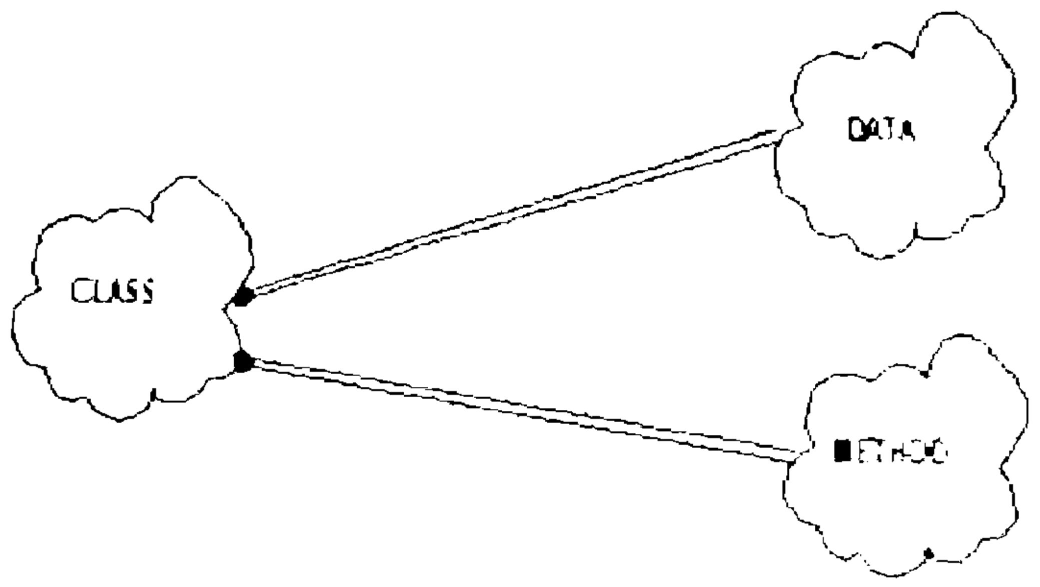
Interface of a class provide its outside view and therefore emphasizes the abstraction while hiding its structure and its behavior. This interface primarily consists of the declaration of all the operations applicable to instances of this class, but may also include declaration of constants, variable and exceptions as needed to complete the abstraction.

The *implementation* of a class primarily consists of the implementation of all the operations defined in the interface of the class.

Thus fundamentally the class should capture the above ideas through information about data and methods together with cardinality, constraints, nature of access for other classes. As design evolves this can be refined by adding details like concurrency, space complexity.

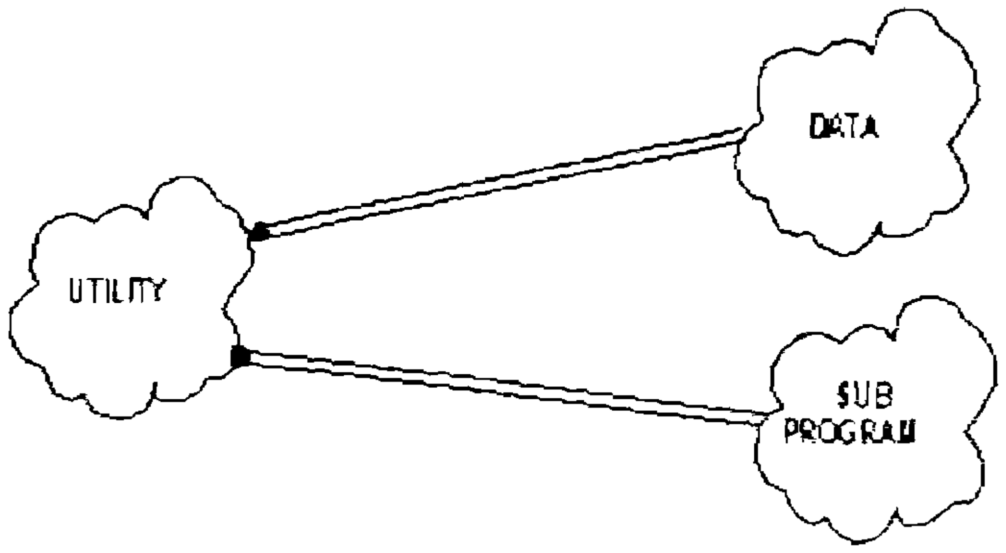
The methods that are captured as part of semantics of the class are designed to capture the usual details like parameters(formal, return), advanced details like time complexity, space complexity etc.,

'Class' object contains data and method objects as shown



Class utility:

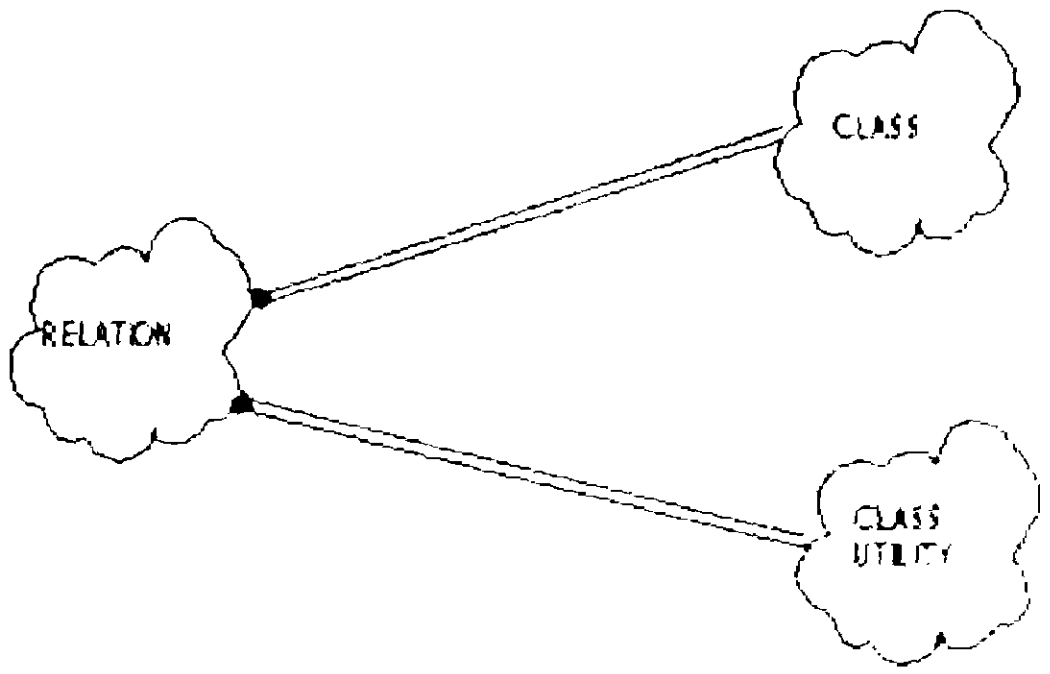
Class utility denote collection of *free subprograms*(non member functions in the system--legacy from the procedural programming style) together with some common data members. ***Free subprograms*** provide some common algorithmic services built upon two disparate lower-level abstractions. Rather than associating these operations with a higher-level class, we choose to collect them in a class utility to increase their chance of reuse, because this provides a finer granularity of abstraction.



Relationships among the classes:

Classes are usually related in a variety of interesting ways, forming the class structure of our design. We establish relationships between two classes for one of the reasons

- A class relationship might indicate some sort of sharing.
- A class relationship might indicate some kind of semantic condition.



1. Association:

Association denotes a ***semantic dependency*** and does not state either the direction or the exact way in which one class relates to another. As we continue our design and implementation, we will often refine these weak associations by turning them into one of the other more concrete class relationships. Multiplicity is attached at each end to signify the cardinality across an association. This is allowed from class/class utility to class/class utility.

2. Inheritance:

Simply stated, inheritance is a relationship among classes, wherein one class shares the structure and/or behaviour defined in one or more classes.

3. Aggregation(Has):

An aggregation relationship corresponds to whole or part hierarchy. As it is containment relationship this is allowed from class/class utility to class.

4. Using:

A using relationship among classes parallels the peer-to-peer links among the corresponding instances of these classes, whereas an association denotes a bi-directional semantic connection. A using relation is one possible refinement of an association. Typically, a using relation manifests itself by the implementation of some operation declarations. This is allowed from class utility/class to class class utility.

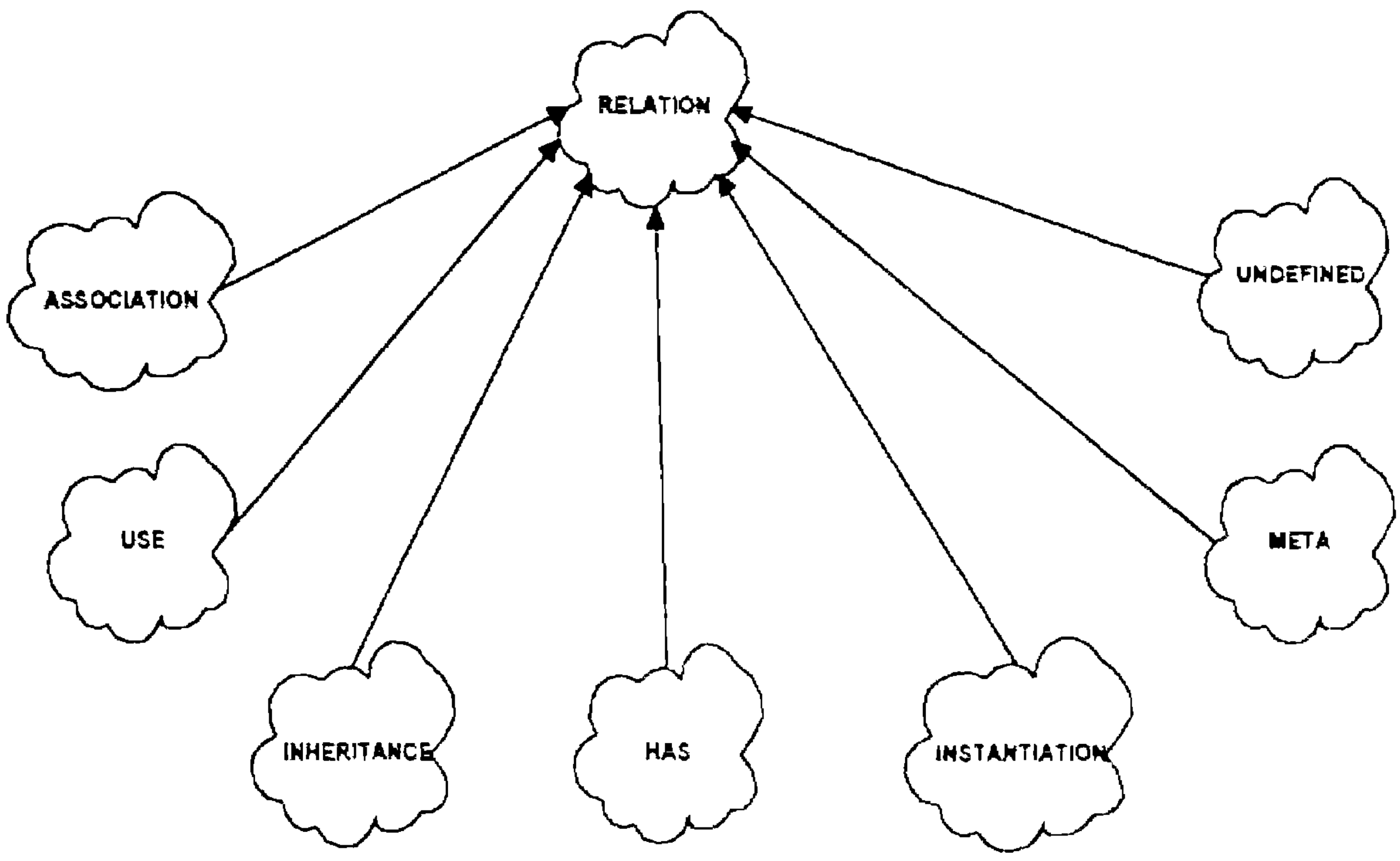
5. Instantiation:

This corresponds to template facility in C++. A parameterised class cannot have instances unless we first instantiate it

6. Meta:

Meta class is a class whose instances are themselves classes. Robson observes " In a system under development a class provides an interface for the programmer to interface with the definition of objects "[8].

All the above relations share some common properties like, relations are between some entities, are attached with some responsibilities etc.,



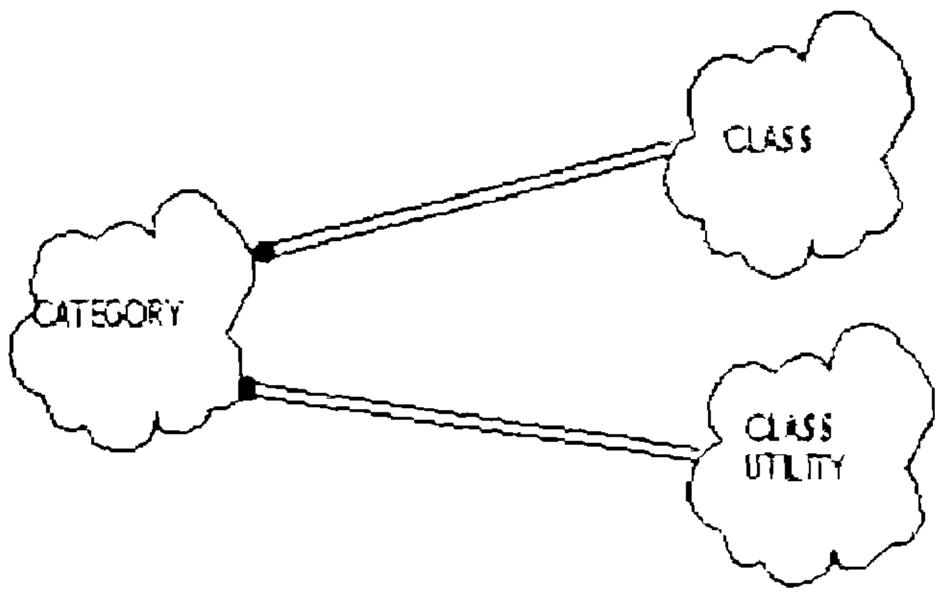
Class Categories:

A class category is an aggregate containing classes and other class categories. Each class in the system must live in a single class category or at the top level of a system. Unlike class, category does not directly contribute state or operation to the model, it does so only indirectly through its contained classes.

Class categories serve to partition the logical model of the system. These categories are also known as subsystems, subject areas, modules in other methodologies other than Booch.

Some of the classes enclosed by a class category may be public, meaning that they are exported from the category and hence usable outside the class category. Other classes may be private, meaning that they are not usable by any other class outside the category.

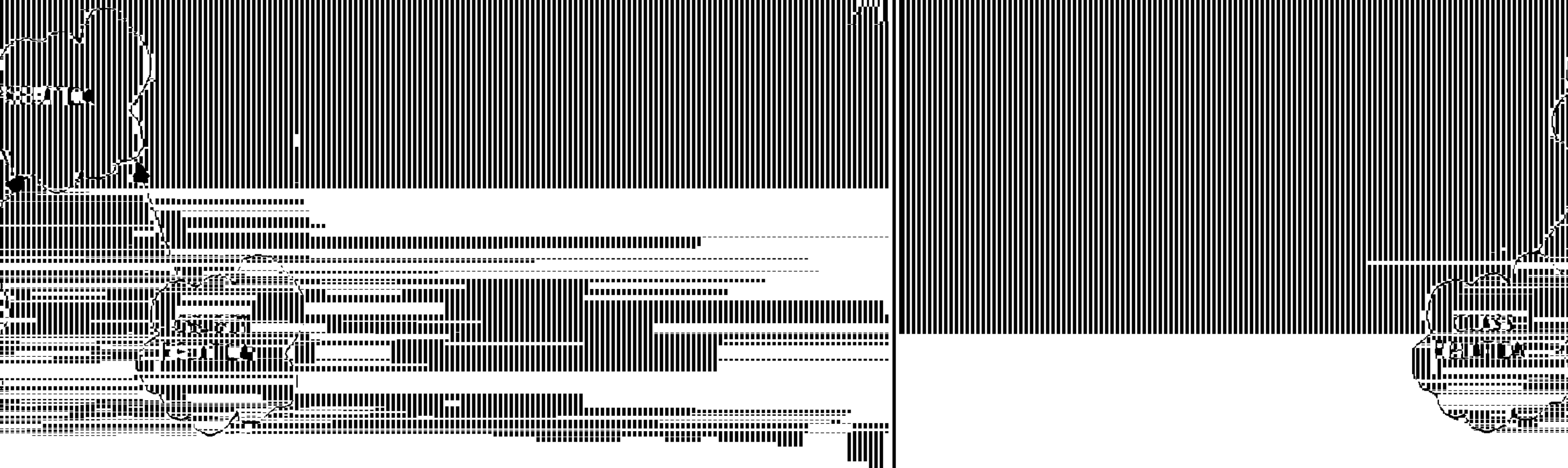
But sometimes, a category containing some classes may need to be used by many categories in the system(E.g., application framework classes). To denote that, global attribute is useful.



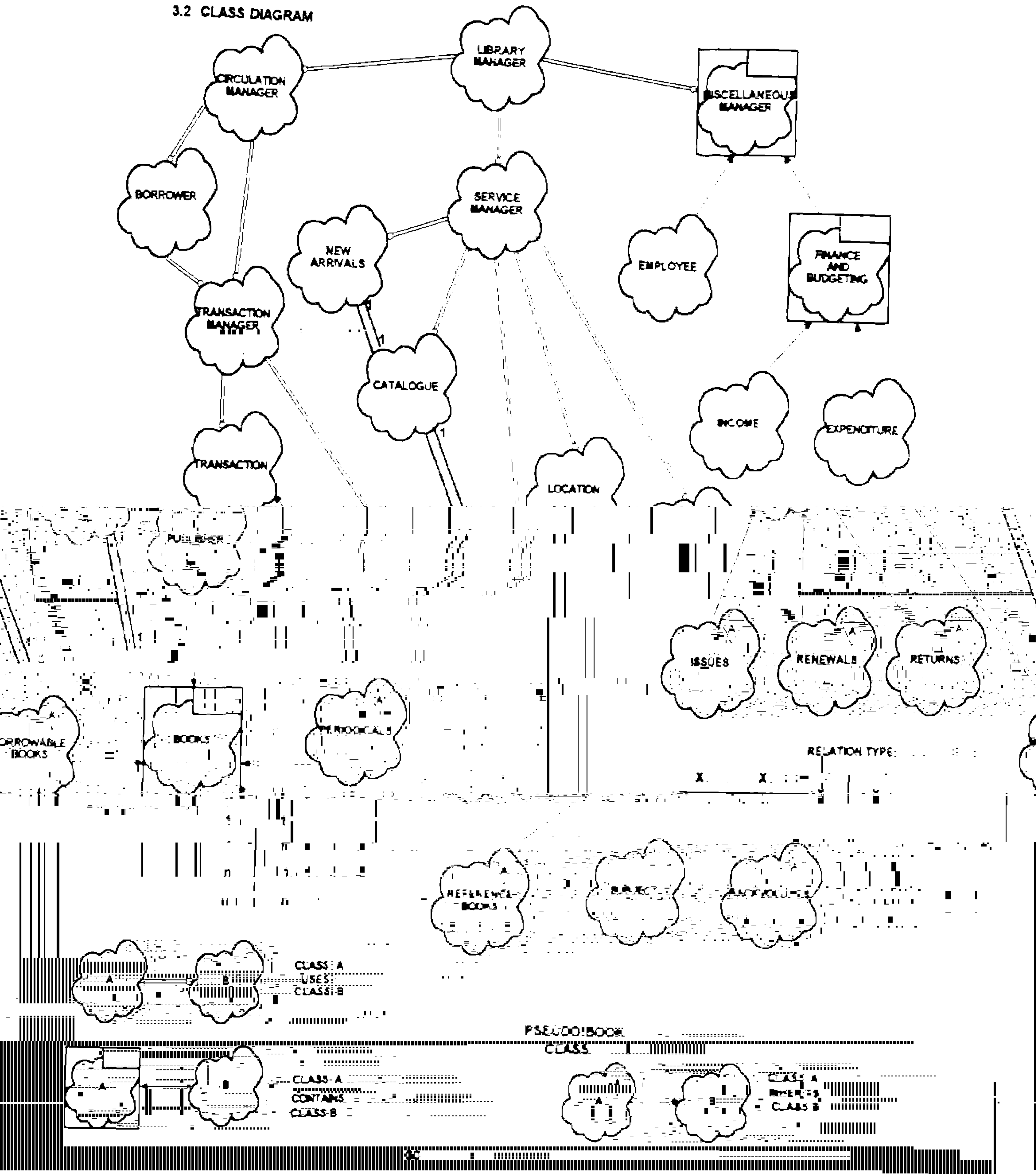
Relationships Among Categories:

Using:

If one or more class/class utility in a category uses one or more classes in another category, it is represented through the using relationship.



3.2 CLASS DIAGRAM



3.3 Description of the design

Let us walk through the design and see how the essential features of the library are captured in the Object Model. A descriptive explanation follows:

1. Books Class:

This Class can be thought of as the heart of the entire design. The entire library operations are centered around this Class. Borrowable Books, Reference Books, Periodicals and Back volumes are all basically Books. They have certain properties like Book-Number, Title, Author(s), Price etc., in common. These commonalities are captured in the Base Class called Books.

Each Book speaks about a particular subject(ideally). Hence each Book contains an instance(containership) of Subject Class (to be elaborated later).

Also every Book has to be published by some publisher. Hence the Books Class also contains an instance of Publisher Class(explained later).

This Class contains Virtual Functions(explained in chapter 4) that execute different functions in different classes for the same function call

2. Borrowable Books Class:

This class is derived from base class Books. It has all the qualities of books and adds it's own extensions, for e g , Borrowable books can be taken out of library, have fine for dereliction of due dates , date of issue, renewal dates etc.,

3. Reference Books Class:

This class is derived (or inherited) from the base class Books. It has all the qualities of books and adds its own extensions (both data and functions) to the base class. For e.g. Reference books are books that cannot be taken out of the library, have hours allotted for referring them etc.,

4. Periodicals Class:

This class is derived (or inherited) from the base class Books. It has all qualities that a book can have and adds its own extensions(both data and functions).Periodicals are books that have periodicity, Issue-nr. Issue-date, Subscription Amount, Subscription-period etc.,

5. Back volumes Class:

The back volume class is derived from the Books class. Back volumes are books and share some common qualities with periodicals. Further they have their own features like number-of-issues present in that back volume, starting and ending dates of issues in that back volume etc.,

6. Publisher Class:

The publisher class is used to capture the publisher information. Normally libraries buy books by placing order to publishers. So the information pertaining to the publisher is stored through this class template. If a person wants to know the set of all publishers from whom the library orders he/she can get it so from this class. Also information about who published a particular book etc., can also be had.

An Instance of this class is contained within the books class.

7. Subject Class:

The subject class is used to capture subject information. Each book is about a particular subject and sometimes with a sub area. For example a book can be on computer science and deal about Databases (a sub area of computer science). All this information are stored using the subject template.

An Instance of this class is contained within the books class.

Note:

The classes that are enclosed within dotted lines are thought of as the technical services offered by the library. (see the design in Sec 3.2).

8. Location:

The location class is used to help user's in locating a particular book in the library. Each book can be kept in only one place (even though there may be multiple copies each copy can occupy one place only).So there is a **one to one relationship** between Books and Location Class.

By entering books information (or what all he knows, say, it may be Book-nr or Title or etc.,) and pressing a key he can get the location information.

9. New arrivals class:

Libraries normally order for books from publishers. The books placed in the order are delivered to them. To capture this **acquisition** process the new arrivals class is used. The service manager class (to be explained later) calls on the new arrivals class to keep track of new Books acquired by the library.

Information like Invoice number, Publisher particulars, Date of placing the orders and receiving the books, list of books etc., are some of the fields in the class. If a person wants to know the set of all books newly bought before or after a particular date, latest arrivals from a particular publisher etc., he can get those information from the new arrivals class.

10. Catalogue class:

As told in the Sec 2.3 catalogue is like a guide. It helps a user in easily locating the book he wants. Every book has a catalogue associated with it. So there is a one-to-one relationship between books and catalogue. At the press of a key the Catalogue information of a book can be had.

11. Service Manager:

The service manager class is incharge of the services offered in the library. It updates and keep consistent data about Books, Publishers, Location, New arrivals and Catalogues. A user can get information from Service Manager, by telling his query, which calls upon the other classes by passing and receiving messages.

The service Manager also contains information about To members served, Total no-of-books, Total-no-of-periodicals, Total-c of-books, No-of-publishers etc., to name a few of the attributes in class.

12. Borrower Class:

Borrowable books are taken from the library by borrow Borrower information like Borrower-number, Name, Professi Address, No-of-cards, Cards availed, List of books held etc., are stored using the Borrower class template.

The Borrower transacts with the library through transaction manager to acquire borrowable books (subject to availability). Information regarding a particular Borrower who has borrowed a particular book, fines to be paid by the borrower etc., are all available through this class.

13. Transaction:

. Normally users of the library make transactions like Issues, Renewals and Returns. Basically borrowing of books , renewal of bo and return of books are all transactions. They have some comm features like Transaction-nr, Borrower-nr, Transaction-date, Transac type(Issue/return/renewal) Books-involved etc., in the transaction. these details are captured using the Base class called Transaction.

Transaction class consists of virtual functions that exe different functions in Issues, Returns, Renewals class for the s function calls.

14. Issues Class:

This class is derived from the Transaction class. Issue transaction in the library, so it inherits all the qualities of the base c and adds its own extensions like Issuing-clerk-nr, date-of-issuing, numbers upon which issue is done etc.,

15. Returns Class:

This class is derived (or inherited) from the Transaction cl Return is a transaction in the library. So it inherits all the features transaction and adds its own features like return-clerk-nr, due-date return, date-of-actual-return, fine collected for delay in returning ,car be returned etc.,.

16. Renewals Class:

This class is derived (or inherited) from the transaction class. Renewal is a transaction in the library. It has all the characteristics of a Transaction and adds its own like actual-date-to-renew, date-of renewal, fine-due-to-late-renewal etc., to the Base Class.

17. Transaction Manager:

The Transaction manager class supervises all the transactions in the library and maintains orderliness. Information regarding a particular transaction, transaction on a particular date, transaction made by a particular clerk etc., can be had by interacting with this class. The Transaction manager also sees whether a particular book is available or not when requested by borrowers and if so issues it.(if requested)

18. Circulation Manager:

The entire circulation operation of the library like Issues, Returns, Renewals, keeping track of Borrower information etc., are all looked after by this class. The circulation manager calls appropriate classes based upon the queries given. If a person wants borrower information , borrower class is called , or if a person wants transaction information , transaction class is called and so on.

Further this class contains attributes like, total number of borrowers, average books issued daily, average enquiries-daily, average visitors-per-day , average returns , renewals, issues per day, fines collected etc.,

19. Employee Class:

The employee class serves as a template to store information about the employees present in the library. This class can be used as base class if a library wants to differentiate people into workers, assistants, librarians, deputy librarians, head librarian etc.,

Employee information like Employee number, Name, Designation, Pay, Duties are stored using this class as template.

20. Income class:

The income class is used to capture the information about the revenues to the library. A library may get funds through state or central govt, endowments, fines and fees, interests etc.,. The date of getting income, from whom, by which media (cheques, cash, DD etc.,) are all the attributes in this class.

21. Expenditure class:

Similar to the Income class, the library allocates its funds in different ways like buying Books, giving Salaries and Wages, paying Electricity charges, Maintenance charges, Subscription to News papers etc., .All these information are stored using this class template.

22. Finance and Budgeting class:

This class which is a container class of Income and Expenditure deals with the financial operations of the library

23. Miscellaneous Manager class:

From the chapter 2.3 we saw that personnel administration and finance come under the Miscellaneous operations in a library. All the operations under this banner are looked after by this class. It is a container class of Employee class and Finance and Budgeting class

24. Library Manager Class:

This class can be told as the head of the design. This class is incharge of the entire library operations. It is incharge of maintaining data that is complete, consistent and reasonable about the library. The user interface is also stored in this class to make the design hierarchical.

This class also contains information general to library like name of the library, nature of the library, area occupied, population served, working hours, days closed, type of service offered etc.,.

Chapter 4

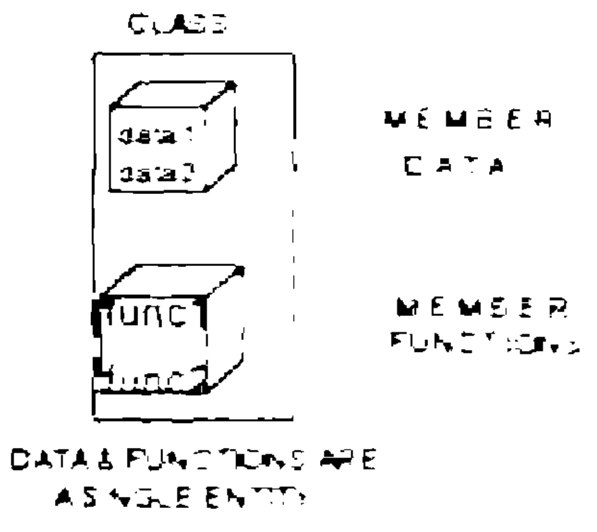
IMPLEMENTATION

4.1 Concepts involved in the implementation

As mentioned in earlier chapters the various features of OO programming have been exploited to achieve the goal. Now, let us see where and how in the design and implementation these concepts are being encompassed.

Objects & Classes:

An Object is said to be an instance of a Class, in the same way as chevrolet is an instance of a vehicle. A Class is an abstraction that consists of data items and functions which capture the structure and behavior common to all related objects. The data members of a class can be accessed from outside using only the member functions in that Class. Placing data and functions together into a single entity is the central idea of Object-Oriented programming.

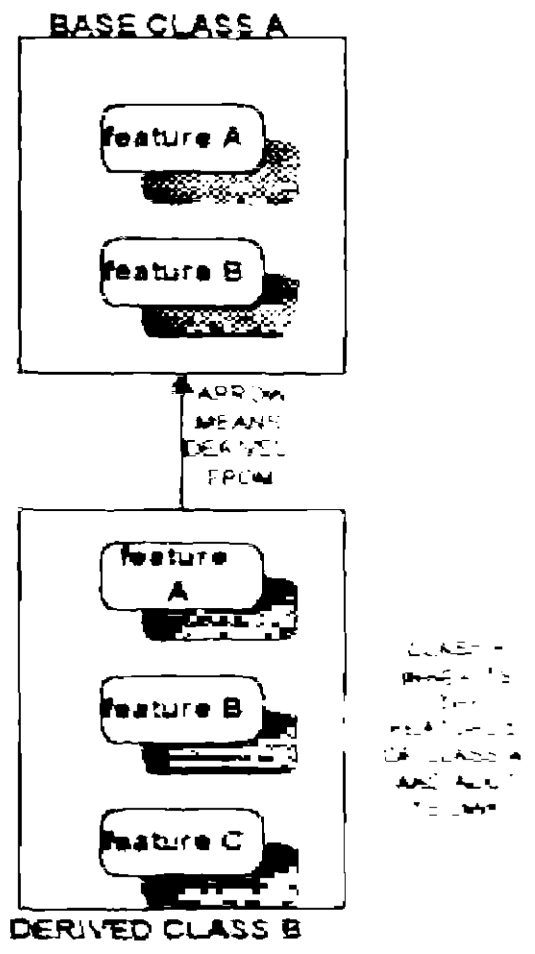


In Our Design and Implementation:

The various classes that are present in our design have been presented in chapter 3.2. The Implementation details are given in chapter 4.2 .

Inheritance:

A Class, called the derived class, can inherit the features of another class, called the base class. The derived class can add other features of its own, so it becomes a specialized version of the base class. Inheritance provides a powerful way to extend the capabilities of existing classes, and to design programs using a hierarchical approach.



For the situation shown in the diagram the syntax will be,

```
Class A
        {
         Feature A;       // Base Class A
         Feature B;
        };
Class B: Public A
        {
         Feature C;       // Derived Class B
        };
```

Access Specifiers:

An important topic in inheritance is knowing what the derived class(es) can (data and functions) access from their base class(es). There are a whole raft of possibilities depending upon the keywords used.

Access Specifier	Accessible from Own Class	Accessible from derived Class	Accessible from Objects outside Class
Public	Yes	Yes	Yes
Protected	Yes	Yes	No
Private	Yes	No	No

Note:

Class Members(which can be data or functions) can always be accessed by functions within the own class, whether the members are *private* or *protected*. But Objects of a Class defined outside the Class can access class members only if the members are public

A *Protected* member, on the other hand, can be accessed by member functions in its own class or- here's the key- in any class derived from its own Class. It can't be accessed from functions outside the classes such as main( ).

When we are writing a class that we suspect might be used, at any point in the future, as a Base class, for other classes, then any data or functions that the derived classes might need to access should be made *protected* rather than *private*. This ensures that the Class is "**inheritance ready**".(to adapt a phrase from TV set advertising).

In Our Design and Implementation:

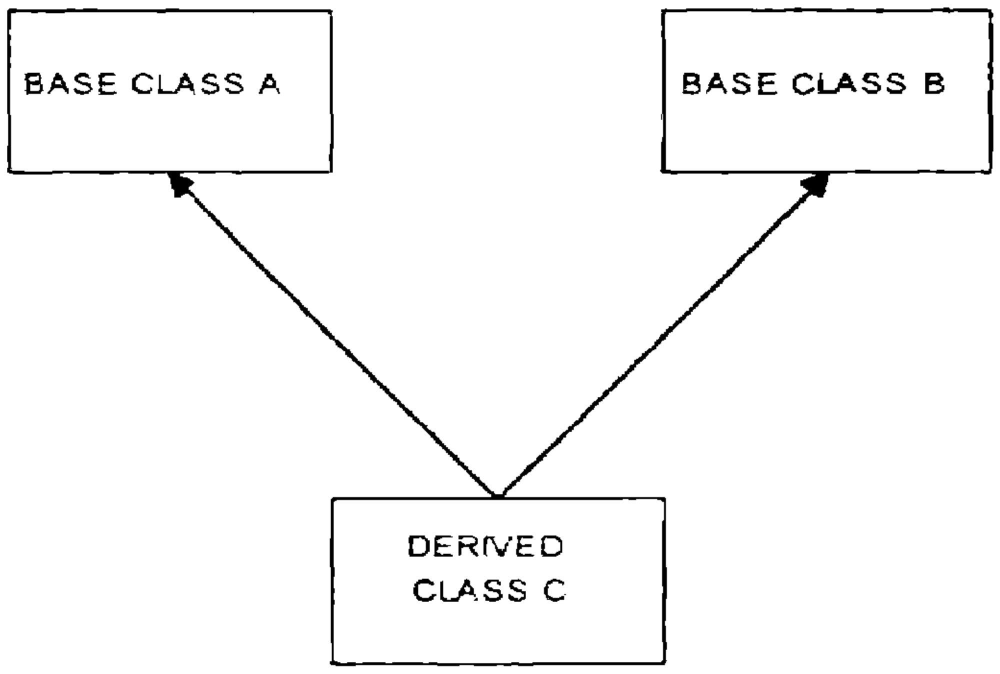
Inheritance is used at the very heart of our design. One can see that Borrowable Books, Reference Books, Periodicals, Back volumes are all basically Books. So a base class capturing the essential qualities of Books like Number, Title, Author, Price, Edition etc., is defined. All the other books are derived from this Base class and add their own extensions say Borrowable Books have some fine on them, Reference Books cannot be borrowed, Periodicals come in stipulated time periods, have subscriptions and so on--- to name a few of them.

Furthermore see the Issues, Returns and Renewals classes. All these three are Transactions made by the borrower with the library. There are some basic qualities for each of these transactions like Transaction-date, Transaction number, Borrower number, Books
volved in the Transaction and so on. So a Base class called
ansaction to incarcerate these commonalities is defined and the other
ree classes are derived from it. The derived classes have their own
ntributions to make, say, Issues have the Issue-date of the book, Due-
te for return or renewal(as the case may be), Returns and Renewals
ve fine involved for dereliction of duty on the part of the Borrower
c., .

From these descriptions we can see, how Inheritance concept is being taken advantage of. The implementation details of inheritance are given in chapter 4.2 .

Multiple Inheritance:

A Class can be derived from more than one Base class. This is called multiple inheritance. This is shown in the diagram where a Class C is derived from the base classes A and B.



The syntax for multiple inheritance is similar to that of single inheritance. For the situation shown in the figure, the relationship is expressed like as shown below

```
Class A
    {
    member data and functions.
    };
Class B
    {
    member data and functions.
    };
```

```
Class C: Public A, Public B
        {
                        //Class C derived from A and B
        };
```

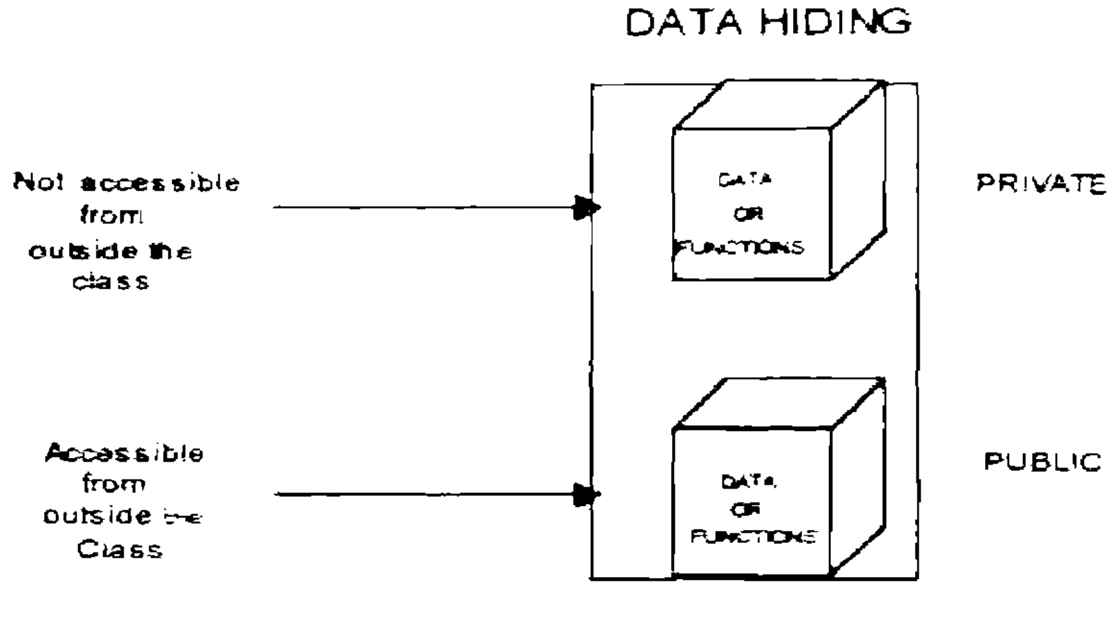
<u>In our Design and Implementation</u>:

Multiple Inheritance can be used in our design and implementation if the library wants to differentiate employees into Head Librarians, Deputy Librarians, workers etc., .

Data Hiding & Encapsulation:

The above term does not refer to the activities of particularly paranoid programmers; rather it means that data is concealed within a class, so that it cannot be accessed mistakenly by functions outside the class. Their primary mechanism for hiding data is to put it in a class and make it private. Private data or functions can only be accessed from within the class. Public data or functions, on the other hand, are accessible from outside the class.

This is shown in the figure.


DATA HIDING
Not accessible from outside the class
DATA OR FUNCTIONS
PRIVATE
Accessible from outside the Class
DATA OR FUNCTIONS
PUBLIC

One should not confuse data hiding with the security techniques used to protect computer databases. To provide a security we might, for example, require a user to supply a password before granting access to a database. The password is meant to keep unauthorized or malevolent users from altering (or often even reading) the data.

Data Hiding, on the other hand, is designed to protect well-intentioned programmers from honest mistakes. Programmers who really want to can figure out a way to access private data, but they will find it hard to do so by accident.

In Our Design and Implementation:

The above concept is involved in all the class definitions.

Containership:

In inheritance, if a class B is derived from a class A, we can say that :"B is a kind of A". This is because B has all the characteristics of A, and in addition some of its own. It's like saying that a Starling is a kind of bird: A Starling has the characteristics shared by all birds(wings, feathers and so on) but has some distinctive characteristics of its own(such as dark iridescent plumage). For this reason inheritance is something called a "kind of" relationship.

There's another kind of relationship, called a "has a" relationship, or containership. We say that a Starling has a tail, meaning that each Starling includes an instance of a tail. In OOP, the "has a" relationship occurs when one object is contained in another. Here's a case where an object of class A is contained in a class B.

```
Class A
        {

        };

Class B
        {
        A a; // a is an object of Class A.
        };
```

In Our Design and Implementation:

Containership feature of OOP is usurped in our design at two pivotal places.

One is,

Every Book has to be published by a publisher(ideally) and talks about a particular subject. So in the Books class there will be an instance of publisher class and Subject class contained.

```
Class Book
        {
         Book-Nr;
         Title;
          ------
         Publisher P;
         Subject S;
         ------
         Price;
```

As given above, one can see that an object of Publisher class and Subject class is contained within the class Book giving rise to containership.

Trekking into the design further we can see containership in another place also. The Miscellaneous Manager Class is a container class of Employee class and Finance and Budgeting class.

```
Class Miscellaneous-Manager
                              {
                               Employee emp;
                               Finance fin;
                              };
```

But it doesn't end here, the Finance & Budgeting Class itself contains Income and Expenditure classes as its two eye s(so as to tell it's importance).

```
Class Finance
          {
           Income inc;
           Expenditure exp;
          };
```

Containership is clearly useful with classes that act like a data type.

Constructors:

Automatic Initialization of objects of a particular class can be done using member functions called Constructors. These special member functions are executed whenever an object of that class is created. There can be more than one constructor in a class depending upon how initialization is to be done.

For example,

```
                    Class Counter
                            {
                            private:
                            int count;
                            Public:
                            Counter( )
                                    {
                                      count=0;
                                    };
                            }
```

defines a constructor called Counter that initializes count to zero. The only restriction is that the Constructor name should be same as that of Class declaration. Moreover Constructors do not return any value.

Constructors are pretty amazing, when you think about it. Whoever writes language compiler(be it for C or Basic or even C++) must execute the equivalent of a constructor when user defines a variable. If you define an *int*, for example, somewhere there's a constructor allocating 2 bytes for it.

Destructors:

We've seen a special member function- the constructor - is called automatically when an object is first created. One might guess that another function is called automatically when an object is destroyed. This is indeed the case. Such a function is called a Destructor. A Destructor has the same name as the constructor(which has the same name as the constructor, which is the class name) but preceded by a *tilde*(~).

In last example it would be

```
{
  Public:
   Counter() { count = 0 }; // constructor
  ~Counter() {};            // destructor
}
```

Like constructors, destructors do not have a return value. They also take no arguments. The most common use of destructors is to deallocate memory that was allocated for the object by the constructor.

In Our Implementation:

Quite a few *constructors* and *destructors* have been used in our implementation,the details of which are given in Chapter 4.2.

Virtual Functions:

Virtual means ***existing in effect but not in reality***. A Virtual function, then, is one that does not really exist but nevertheless appears real to some parts of a program.

Why are virtual Functions needed? Suppose you have a number of objects of different classes but you want to put them all on a list and perform a particular operation on them using the same function call. For e.g., suppose a graphics program includes several different shapes, a triangle, a ball, a square and so on. Each of these classes has a member function draw( ) that causes the object to be drawn on the screen.

Now suppose you plan to make a picture by grouping a number of these elements together, and you want to draw a picture in a convenient way. One approach is to create an array that holds pointers to all the different objects in the picture. The array might be defined like this

```
Shape * Ptrarr[100];        // array of 100 pointers to shapes.
```

If you insert pointers to all the shapes into this array, you can then draw an entire picture using a simple loop;

```
for(int j=0; j < N; j++)
        ptrarr[j]->draw( );
```

This is an amazing capability; completely different functions are executed by the same function call. If the pointer in Ptrarr points to a ball, the function that draws a ball is called; if it points to a triangle, the triangle-drawing function is called. This is an important example of ***polymorphism*** or giving ***different names to same thing.***

But for this Polymorphic approach to work, several conditions must be met. First of all, the different classes of Shapes, such as balls and triangles, must be derived from a single base class. Second the draw( ) function must be declared to be Virtual in the base class.

For e.g., if Shape was to be the base class from which all the other classes (square, triangle, etc.,) are derived, then in Shape class we should define draw( ) as below,

```
Virtual Return-type draw( );
```

In Our Implementation:

In our implementation virtual functions are present at places where inheritance has been used. After all both of them go hand in hand. Don't they?.

Reference Books, Borrowable Books etc., are derived from Base class Books satisfying the first requirement of the previous para. Then within this Base class functions like Showbook( ), Modifybook( ) etc., are declared Virtual thereby satisfying the second requirement.

Virtual functions are facilities provided by the language compiler by deferring certain decisions until run time. At run time, the appropriate version of the function to be executed is found out. This is called ***Late Binding*** or ***Dynamic Binding*** (choosing functions in the normal way, during compilation, is called early, or static binding). Late binding requires some overheads but provides increased power and flexibility.

Operator Overloading:

Operator Overloading is one of the most exciting features of Object Oriented Programming. It can transform complex, obscure program listings into intitutively obvious ones.

For example, a statement like

d3.addobjects(d1,d2);

can be changed to the much more readable,

d3 = d1 + d2 ;

The rather forbidding term operator overloading refers to giving the normal C++ operators, such as +,-,*,<=,+= etc., additional meanings when they are applied to user defined data types.

In Operator Overloading, that involves conversions of user defined types, the programmer has to do some part by giving new definitions for the operators. For example if the ++ operator has to be redefined for the user application then he has to do so as follows:

```
Void Operator ++ ( )
          {
            Programmer defined statements to be
            executed for ++ operator;
          };
```

Here **operator** is the keyword that is to be used to overload operators.

Overloaded Operators are not all Beer and skittles even though they give high flexibility. They have their own drawbacks as if too many are used the listing may not be comprehensive, thereby defying their very use.

In Our Implementation:

On experimental basis the = (assignment operator) was overloaded to assign a book object to another book object.(e.g., b1 = b2 where b1 and b2 are objects of type Books).

Overloaded functions:

An overloaded function appears to perform different activities depending upon the kind of data sent to it. Overloading is like the joke about the famous scientist who insisted that Thermos Bottle was the greatest invention of all times. Why? " It is a miracle device," he said. " It keeps hot things hot, but cold things it keeps cold. How does it know?".

It may seem equally mysterious how an overloaded function knows what to do. It performs one operation on one kind of data but another operation on a different kind.

For example, consider the following segment of listing

```
void repchar( );
void repchar(char );
void repchar(char , int );
void main( )
          {
            repchar( );
            repchar( '=');
            repchar('+ ', 30);
          };
```

and three repchar( ) functions each doing different operations. Say one repchar prints 45 asterisks, the next prints 45 times the char given as argument, and the final one prints the char given int times.

The compiler on seeing several functions with same name but different number of arguments, could decide the programmer had made a mistake. Instead, it very patiently sets up a separate function for every such definition. Which one of the function will be called depends upon the number of arguments supplied in the call.

In Our Implementation:

The various Overloaded functions that are present in our implementation are given in Chapter 4.2. To name a few of them the Modify_Catalogue( ) function in the Catalogue Class, Show_publisher( ) in the Publisher Class, Show_Location( ) in Location class etc., .

Inline functions:

An inline function looks like a normal function in the source file but inserts the function's code directly into the calling program. Inline functions execute faster but may require more memory than normal functions unless they are small.

The keyword **inline** is used to specify a function as inline.

<u>In Our Implementation:</u>

Only one or two Inline functions have been used. For example the function that calculates fine to be paid by a borrower is an *Inline* function, the details of which are given in chapter 4.2.

Friend Functions:

The concepts of encapsulation and data hiding dictate that non-member functions should not be able to access an Object's private or protected data. The policy is, if you're not a member, you can't get in. However, there are situations where such rigid discrimination leads to considerable inconvenience.

For example, if you want a function to operate on objects of two different classes. Perhaps the function will take objects of the two classes as arguments, and operate on their private data. If the two classes are inherited from the same base class, then you may be able to put the functions in the base class. But what if the classes are unrelated?

In this situation there's nothing like a friend function. A friend function acts as a bridge between the two classes.

To access the private data in both the classes, say Class A and Class B, the function should be declared as friend within both the classes, as shown below *friend return-type funcname*(A a , B b);

where a and b are objects of type Class A and B respectively.

Similarly if Class A must access Class B's private data then, within Class B we should declare Class A as friend class. This is shown below

friend class Class A;

In Our Implementation:

The details of various friend functions and classes used in our implementation are given in chapter 4.2.

Persistence:

Persistence is essential for information to reside on the permanent memory. The information stored in the disk needs to be read and written to satisfy queries, to make modifications etc.,. File Streams of Borland C++, is used to achieve the purpose of persistence.

Borland C++ provides a wealth of classes to achieve persistence. Classes are arranged in the Class Library in a rather complex hierarchy. We've made extensive use of these classes. For example **ifstream**, **ofstream** and **fstream** classes defined in the FSTREAM.H header file.

The class **ifstream** is used for input operations on files, **ofstream** is used for output operations on files and **fstream** for files that will be used for both input and output operations. Reading and writing of information from the storage medium is done in the form of Objects.

In Our Implementation:

As told earlier persistence is achieved using disk files. The various files used in our implementation are given in chapter 4.2.

4.2 Implementation details

4.2.1 Class LibraryManager

```
{
Private :
        char * Name_of_the_library;
        char * Nature_of_the_library;
        date day_established; //date is mm/dd/yy
        float Area;
        int Population_served;
        int Working_hours;
        int Days_closed;
        Struct Miscellaneous
                {
                int No_of_xerox;
                int No-of_racks;
                char *publications;
                int No_of_coolers;
                };
```

Public :

1. *void LibraryManager( )*

Purpose:

This member function is a constructor and is used to assign general information about the library.

2. *void ShowDetails( )*

Purpose:

This member function is used to show general information about the Library.

3.*LibraryManager ModifyDetails( )*

Purpose:

To modify the general information of the library, say a new cooler could have been bought or a new set of racks added etc.,this member function is used. It returns the modified object.

4. *void diskinlib*( )

Purpose:

This member function is used to write the information present in this class's object to the disk.

5. *LibraryManager diskoutlib*( )

Purpose:

This member function is used to retrieve the information present in this class's object from the disk.

6. *friend int LibMenu*( )

Purpose:

This declaration specifies that the LibMenu( ), which is the user interface function for OOLIS, is a friend of this class.

7. *~LibraryManager*( )

Purpose:

This member function is a destructor and is used to deallocate memory.

File associated with this class is LibraryManager.dat .

}

4.2.2.3 Class CirculationManager

{

Private:

int No_of_borrowers;

int Ave_enquiries_daily;

int Ave_visitors_daily;

int No_of_books_referred;

```
        struct usage
                {
                int ave_issues_daily;
                int ave_returns_daily;
                int ave_renewals_daily;
                };
        float ave_fines_per_day;
```

public:

1.*void CirculationManager*( )

Purpose:

This member function is a constructor and is used to assign initial values.

2. *void showinfo*( )

Purpose:

This member function is used to display circulation information of the library.

3. *circulationmanager modifyinfo*( )

Purpose:

This member function is used to modify information present in the circulation manager class. It returns the modified circulation manager object.

4. *void diskincirman*( )

Purpose:

This member function is used to write circulation information onto the disk.

5. *circulationmanager diskoutcirman*( )

Purpose:

This member function is used read circulation information from the disk. It returns the read circulation manager object.

6. *friend class LibraryManager*

Purpose:

This declaration specifies that the circulation manager class is a friend of library manager class.

7. *friend int cirmenu( )*

Purpose:

This declaration specifies that the cirmenu( ), which a menu function for circulation operations(borrower operations, transaction operations etc.,), is a friend of circulation manager class.

8. *~CirculationManager( )*

Purpose:

This member function is a destructor and is used to deallocate memory.

File associated with this class is Circulation.dat .

}

4.2.3 Class ServiceManager

{

Private:

```
        int Total_members_served;
        int Total_borrowable_books;
        int Total_periodicals;
        int Total_backvolumes ;
        int Total_Reference_Books;
        int Total_cost_of_books;
        int Total_cost_of_Subscriptions;
        char *Magazines_ordered[ ];
        int No_of_publishers;
```

public

1. *void ServiceManager( )*

purpose:

This member function is a constructor and is used to assign some initial values to the private data.

2. *void ShowInfo( )*

purpose:

This member function is used to display the information contained in the Service manager class.

3. *ServiceManager Modifyinfo( )*

purpose:

This member function is used to modify the information contained in the service manager class. It returns the modified object.

4. *void diskinSer( )*

Purpose:

This member function is used to write the data in Service Manager class on to the disk.

5. *ServiceManager diskoutSer( )*

Purpose:

This member function is used retrieve Service Manager information from the disk.

6.*friend class LibraryManager*

Purpose:

This declaration specifies that Service Manager class is a friend of Library Manager class.

7.*friend int SerMenu( )*

Purpose:

This declaration specifies that SerMenu( ), which is a menu function for Service Manager operations , is a friend of this class.

8. *~ServiceManager( )*

Purpose

The above member function is a destructor and is used to reclaim space.

File associated with this class is ServiceManager.dat .

}

4.2.4 Class TransactionManager

{

Private:

int Clerks_incharge_return[ ];

int Clerks_incharge_renewal[ ];

int Clerks-incharge-issue[ ];

date Transaction_day;

Float Fines_collected_for_the_day;

int No_Transactions_for_the_day;

Public:

1. *Void TransactionManager*( )

Purpose:

This member function is a constructor and is used to assign initial values.

2. *void Showinfo*( )

Purpose:

This member function is used to display information present in the Transaction manager class.

3. *TransactionManager Modifyinfo*( )

Purpose:

This member function is used to modify information present in the transaction manager class and return the modified transaction.

4. *void diskintrans*( )

Purpose:

This member function is used to write information in the transaction manager class on to the disk.

5. *TransactionManager diskouttrans*( )

Purpose:

This member function is used to read information contained in the transaction manager class from the disk.

6.*friend class CirculationManager*

Purpose:

This declaration specifies that Transaction manager class is a friend of circulation manager class.

7.*friend int TranmanMenu*( )

Purpose:

This declaration specifies that TranmanMenu( ), which is a menu function for Transaction Manager operations, is a friend of this class.

8. *Boolean IsAvailable*(int Book_no)

Purpose:

This member function is used to check for the availability of a Book by calling the appropriate function in Borrowable Books class.

9. *void Sendreminder*(Int Borr_no)

Purpose:

This member function is used to send a reminder to a Borrower for return or renewal of Book(s).

10. *void ShowTrans*(int clerk_no, char *clerk_type, date day)

Purpose:

This member function is used to get the Transaction information of the particular clerk for a particular day by calling the appropriate function in the Issues or Returns or Renewals class based on the second and third argument.

File used for storing Transaction Manager information is Transman.dat.

}

4.2.5 Class Transaction

{

Protected:

```
        int  Transaction_no;
        char  *Transaction_type; //issues or returns etc.,
        date  Transaction_date;
        int  Borrower_nr;
        char  *Books_involved[];
```

Public:

1.a. *Virtual void Showtransaction*(int Tran_no) = 0

  b. *Virtual void Showtransaction*(int Borr_no,date day) = 0

Purpose:

These overloaded member functions are pure virtual functions and are used to show the transaction details of a particular transaction.

2. *Virtual void Addtransaction*( ) = 0

Purpose:

This member function is an overloaded pure virtual function and is used to add a new transaction.

3.a. *Virtual void Modifytransaction*(int Tran_no) = 0

  b. *Virtual void Modifytransaction*(int Borr_no.date day) = 0

Purpose:

These overloaded member functions are pure virtual functions and are used to modify a transaction.

saction(int Tran_no) = 0

4. *Virtual void Deletetran*

Purpose:

This member function is a pure virtual function and is used to delete a transaction.

Note:

The appropriate function in the derived classes(Issues, Returns, Renewals) are called based upon the value in the Transaction_type field.

5. *virtual void diskintran*( ) = 0

Purpose:

This member function is a pure virtual function and is used to write transaction information onto the disk.

6. *virtual void diskouttran*( ) = 0

Purpose:

This member function is a pure virtual function and is used to read transaction information from the disk.

7. *friend int Tranmenu*( )

Purpose:

This declaration specifies that the Tranmenu( ), which is a menu function that gives options to the user to enter the type of transaction needed (issues/returns/renewals) as the case may be and returns the type of transaction, is a friend of Transaction class.

8. *friend class TransactionManager*

Purpose:

This declaration specifies that the Transaction class is a friend of Transaction Manager class.

}

4.2.6 Class Issues : Public Transaction

{

Private:

date Issue_date;

int Issue_clerk_no;

int Card no[ ];

date date_to_return;

date date_to_renew;

Public:

1.a. *void Showtransaction*(int Tran_no)

b. *void Showtransaction*(int Borr_no,date day)

Purpose:

These overloaded member functions are used to show the transaction details of a particular Issues.

2. *void Addtransaction*( )

Purpose:

This member function is used to add a new Issues transaction.

3.a. *void Modifytransaction*(int Tran_no)

b. *void Modifytransaction*(int Borr_no,date day)

Purpose:

These overloaded member functions are used to modify an Issues transaction.

4. *void Deletetransaction*(int Tran_no)

Purpose:

This member function is used to delete an Issues transaction.

5. *void diskinissue*( )

Purpose:

This member function is used to write Issues information onto the disk.

6.*Issues diskoutissue*( )

Purpose:

This member function is used to read Issues information from the disk. It returns the Issues object.

7. *friend int Issuesmenu*( )

Purpose

This declaration specifies that the Issuesmenu(), which is a menu function for Issues operations, is a friend of Issues class.

8. *friend class TransactionManager*

Purpose:

This declaration specifies that Issues class is a friend of Transaction Manager class.

File associated with this class is Issues.dat

};

4.2.7 Class Renewals : Public Transaction

{

Private:

```
        date  Due_date_of_renewal;
        int   Renewal_clerk_no;
        date  Date_renewed;
        float Fine_collected;
```

Public:

1.a. *void Showtransaction*(int Tran_no)

b. *void Showtransaction*(int Borr_no,date day)

Purpose:

These overloaded member functions are used to show the transaction details of a particular Renewal.

2 *void Addtransaction*( )

Purpose:

This member function is used to add a new Renewals transaction.

3 a. *void Modifytransaction*(int Tran_no)

b. *void Modifytransaction*(int Borr_no,date day)

Purpose

member functions are used to modify a

nt Tran_no)

on is used to delete a Renewal transaction.

on is used to write renewals information on to

)

on is used to read renewals information from
wals object.
)

ecifies that the Renewalmenu( ), which is a
ls operations, is a friend of Renewals class.
t No_of_days, float fine_per_day)

ion defines that fine( ) function, which is an
culate the fine for a book, is a friend of this

Manager

pecifies that Renewals class is a friend of
.

ss is Renewals.dat .

These overloaded
Renewal transaction.
4. *void Deletetransaction*(i
Purpose:

This member functi
5. *void diskinrenew*( )
Purpose:

This member functi
the disk.
6.*Renewal diskoutrenewal*(
Purpose:

This member functi
the disk. It returns the rene
7. *friend int Renewalmenu*(
Purpose:

This declaration sp
menu function for Renewa
8. *friend inline float fine*(in
Purpose:

The above declarat
inline function used to cal
class.
9. *friend class Transaction*
Purpose:

This declaration sp
Transaction Manager class

File associated with this cl

4.2.8 Class Returns : Public Transaction

{

Private:

date Due_date-of_return;

int Returns_clerk_no;

date Returned;

float Fines;

int card_to_be_ret;

Public:

1.a. *void Showtransaction*(int Tran_no)

b. *void Showtransaction*(int Borr_no,date day)

Purpose:

These overloaded member functions are used to show the transaction details of a particular Return.

2.*void Addtransaction*( )

Purpose:

This member function is used to add a new Returns transaction.

3.a. *void Modifytransaction*(int Tran_no)

b. *void Modifytransaction*(int Borr_no,date day)

Purpose:

These overloaded member functions are used to modify a Returns transaction.

4. *void Deletetransaction*(int Tran_no)

Purpose:

This member function is used to delete a Returns transaction.

5. *void diskinreturn*( )

Purpose:

This member function is used to write Returns information onto the disk.

6. *Returns diskoutreturns*( )

Purpose:

This member function is used to read Returns information from the disk. It returns the Returns object.

7. *friend int Returnsmenu*( )

Purpose:

This declaration specifies that the Returnsmenu( ), which is a menu function for Returns operations, is a friend of Returns class.

8. *friend inline float fine*(int No_of_days, float fine_per_day)

Purpose:

The above declaration defines that fine( ) function, which is an inline function used to calculate the fine for a book, is a friend of this class.

9. *friend class TransactionManager*

Purpose:

This declaration specifies that Returns class is a friend of Transaction Manager class.

File associated with this class is Returns.dat .

}

4.2.9 Class Borrower

{

Private:

```
        int Borrower_nr;
        char *Borrower_name;
        char  *Profession;
        char *Address;
```

int No of cards;

int Cards availed;

int Books-held[];// Numbers of Book are stored in this array.

Public:

1. *void Borrower*( )

Purpose:

This member function is a constructor and is used to assign initial values.

2.a. *void showborrower*(int borr_no)

b. *void showborrower*(char *borr_name)

c. *void showborrower*(int book_no)

Purpose:

The above member functions are overloaded functions and are used to show the information about borrower(s) based upon the argument given.

3. *Borrower modifyborrower*(int borr_no)

Purpose:

This member function is used to modify information about a borrower. It returns the modified borrower object.

4. *void deleteborrower*(int borr_no)

Purpose:

This member function is used to delete a borrower.

5.*void sendreminder*(int borr_no)

Purpose:

This member function is used to send a reminder to the borrower regarding books held by him.

6. *void diskinborr*( )

Purpose:

This member function is used to write borrower information onto the disk

7. *Borrower diskoutborr( )*

Purpose:

This member function is used to read borrower information from the disk.

8. *friend int Borrmenu( )*

Purpose:

This declaration specifies that the Borrmenu( ), which is a menu function for borrower operations, is a friend function of this class.

9. *friend class CirculationManager*

Purpose:

This declaration specifies that Borrower class is a friend of Circulation Manager class.

10. *~Borrower( )*

Purpose:

The above member function is a destructor and is used to reclaim space.

File used for persistence is Borrower.dat .

}

4.2.10 Class Newarrivals

{

Private:

```
int  Invoice_nr;
int  Publisher_nr;
date Invoice_date;
date Arrival_date;
float Amount;
```

char *list_of_books[];

Public:

1. *void Newarrivals* ( )

Purpose :

This member function is a constructor and is used to assign initial values.

2.a. *void ShowNewarrival*(int Pub_no)

b. *void ShowNewarrival*(int Invoice_no)

c. *void ShowNewarrival*(date Invoice_date)

d. *void ShowNewarrival*(date from_this_date, int Pub_no)

Purpose:

These member functions are overloaded and are to used to retrieve information about new arrivals based upon the argument.

3.a. *Newarrival ModifyNewarrival*(int Pub_no)

b. *Newarrival ModifyNewarrival*(int Invoice_no)

c. *Newarrival ModifyNewarrival*(date Invoice_date)

Purpose:

[illegible] member functions are overloaded and are used to modify [illegible] information about New arrivals. They return the modified New arrival object.

4. *Newarrival AddNewarrival*()

Purpose:

This member function is used to add New arrivals to the existing ones, and return the newly added New arrival object.

5. *void diskinnew*( )

Purpose:

This member function is used to write the new arrivals information on to the disk.

6. *Newarrivals diskoutnew*()

Purpose:

This member function is used to retrieve the New arrivals information from the disk.

7. *friend int Newmenu( )*

Purpose:

This declaration specifies that Newmenu( ), which is function that presents a menu to the user for New arrivals operations, is a friend of this class.

8. *void Sendreminder(* int Pub_no)

Purpose:

This member function is used to send a reminder to a particular publisher regarding despatch of books.

9. *friend class ServiceManager*

Purpose:

This declaration specifies that New arrivals class is a friend of Service Manager class.

10.*~Newarrivals*( )

Purpose:

This member function is a destructor and is used to reclaim space

File associated with this class is Newarrivals.dat

}

4.2.11 Class Catalogue

{

Private:

```
        float  Call_no;
        int    Acc_no;
        int    Book_no;
        char   *Book_type; (reference/periodical/borrowable etc..)
```

char *Title;

int No_of_copies;

Public:

1. *void Catalogue*( )

Purpose:

This member function is a constructor and is used to assign initial values.

2.a. *void ShowCatalogue*(int Acc_no)

b. *void ShowCatalogue*(float Call_no)

c. *void ShowCatalogue*(char *title)

d. *void ShowCatalogue*(int Book_no)

Purpose:

The above member functions are overloaded and are used to retrieve catalogue information about a book .

3. a. *Catalogue ModifyCatalogue*(int Acc_no)

b. *Catalogue ModifyCatalogue*(float Call_no)

c. *Catalogue ModifyCatalogue*(char *title)

d. *Catalogue ModifyCatalogue*(int Book_no)

Purpose:

The above overloaded member functions are used to modify information about a catalogue.

4. *Catalogue AddCatalogue*( )

Purpose:

This member function is used to add a new catalogue and return the newly added catalogue object.

5. *void diskincat*( )

Purpose:

This member function is used to write the catalogue information onto the disk.

6. *Catalogue diskoutcat( )*

Purpose:

This member function is used to retrieve catalogue information from the disk.

7. *friend class ServiceManager*

Purpose:

This declaration specifies that the Catalogue class is a friend of Service Manager class.

8.a. *void DeleteCatalogue*(int Book_no)

b. *void DeleteCatalogue*(int Acc_no)

c. *void DeleteCatalogue*(char *title)

d. *void DeleteCatalogue*(float call_no)

Purpose:

These overloaded member functions are used to delete a catalogue information..

9. *friend int Catmenu*( )

Purpose:

This declaration specifies that CatMenu( ), which is menu function for catalogue operations, is a friend of this class.

10.~*Catalogue*( )

Purpose:

This member function is a destructor and is used to reclaim space.

File associated with this class is Catalogue.dat .

}

4.2.12 Class Publisher

{

Private:

int Publisher_no;

```
char  *Publisher name;
char  *Address;
int    Telephone_no;
int    Telex;
char   *Cable/gram;
char   *Currency_accepted;
```

Public:

1. *void Publisher*( )

Purpose:

This member function is a constructor and is used to assign initial values.

2.a. *void ShowPublisher*(int Pub_no)

b. *void ShowPublisher*(char *Pub_name)

c. *void ShowPublisher*( )

Purpose:

These overloaded member fun ctions are used to get information about a publisher(s).

3. *Publisher AddPublisher*( )

Purpose:

This member function is used t o add a new publisher, and return the newly added publisher.

4.a. *Publisher ModifyPublisher*(int Pu b_no)

b. *Publisher ModifyPublisher*(char * Pub_name)

Purpose:

These overloaded member fu nctions are used to modify the Particulars of a publisher. They return the modified publisher object.

5. *void DeletePublisher*(int Pub_no)

Purpose:

This member function is used t o delete a publisher.

6. *void diskinpub( )*

Purpose:

This member function is used to write the publisher details on to the disk

7. *Publisher diskoutpub( )*

Purpose:

This member function is used to retrieve publisher details from the disk.

8. *friend int PubMenu( )*

Purpose:

This declaration specifies that PubMenu( ),which is menu function for Publisher operations, is a friend of this class.

9. *friend class ServiceManager*

Purpose:

This declaration specifies that Publisher class is a friend of service manager class.

10. *~Publisher( )*

Purpose:

This member function is a destructor and is used to reclaim space.

File associated with this class is Publisher.dat.

}

4.2.13 Class Location

{

Private:

int Loc_no;

int Book_no;

int floor_no;

int rack_no;

```
        int row_no;
        int column_no;
```

Public:

1. *void Location*( )

Purpose:

This member function is a constructor and is used to assign initial values.

2.a. *void ShowLocation*(int Loc_no)

  b. *void Showlocation*(int Book_no)

Purpose:

These overloaded member functions are used to get the location information of a book.

3.a. *Location Modifylocation*(int Loc_no)

  b. *Location Modifylocation*(int Book_no)

Purpose:

These overloaded member functions are used to modify the location information of a particular book.

4. *void DeleteLocation*(int Loc_no)

Purpose:

This member function is used delete a location information.

5. *void diskinloc*( )

Purpose:

This member function is used write location information on to the disk.

6. *Location diskoutloc*( )

Purpose:

This member is used to retrieve location information from the disk. It returns the location object.

7. *friend int LocMenu*( )

Purpose

This declaration specifies that LocMenu( ),which is a menu function for location operations, is a friend of this class.

8. *friend class ServiceManager*

Purpose:

This declaration specifies that location class is a friend of Service Manager class.

9.*~Location*( )

Purpose:

This member function is a destructor and is used to reclaim the allocated memory.

File associated with this class is Location.dat .

}

4.2.14 Class Subject

{

Private:

```
        int   Subject_no;
        char  *Subject_name;
        char   *Sub_area;
```

Public:

1.*void Subject*( )

Purpose:

This member function is a constructor and is used to assign initial values.

2.a. *void Showsubject*(int sub_no)

b. *void Showsubject*(char *sub_name)

Purpose:

These overloaded member functions are used to retrieve information about a particular subject.

3.a. *Subject Modifysubject*(int sub_no)

b. *Subject Modifysubject*(char *sub_name)

Purpose:

These overloaded member functions are used to modify information about a particular subject. These functions return the modified subject object.

4. *Subject Addsubject*( )

Purpose:

This member function which returns a subject object is used to add a new subject.

5. *void Deletesubject*(int sub_no)

Purpose:

This member function is used to delete a subject.

6. *void diskinsub*( )

Purpose:

This member function is used to write subject information on to the disk.

7. *Subject diskoutsub*( )

Purpose:

This member function is used to retrieve subject information from the disk.

8. *friend int SubMenu*( )

Purpose:

This declaration specifies that SubMenu( ), which is a menu function for subject operations, is a friend of this class.

9. *~Subject*( )

Purpose:

This memb [illegible] used to deallocate space

File associated with [illegible] Subject dat [illegible]

}

4.2.15 Class Books

{

Protected:

```
        int  Acc_no;
        int  Book_no;
        char  *Title;
        char  *authors[];
        Publisher Pub;
        Subject Sub;
        float  Price;
        int  No_of_copies;
        int  No_of_pages;
        char  *Book_type;/ [illegible] periodical etc.,)
```

Public:

1. *void Books*( )

Purpose:

This member functi [illegible] constructor a [illegible] is used to assign initial values.

2.a. *virtual void Showbook* [illegible]

b. *virtual void Showbook* [illegible]

c. *virtual void Showbook* [illegible]

d. *virtual void Showbook* [illegible]

Purpose:

The above overloaded member functions are pure virtual functions and are used to retrieve book information.

3.a. *virtual void Modifybook*(int book_no) = 0

b. *virtual void Modifybook*(int Acc_no) = 0

c. *virtual void Modifybook*(char *title) = 0

d. *virtual void Modifybook*(char *author) = 0

Purpose:

The above overloaded member functions are pure virtual functions and are used to modify book information.

4. *virtual void Addbook*( ) = 0

Purpose:

The above member function is a pure virtual function and is used to add a new book.

5. *virtual void Deletebook*(int Book_no) = 0

Purpose:

The above pure virtual member function is used to delete a book.

Note:

The appropriate function in the derived classes are called based upon the first digit of the Book_no or Acc_no.

6. *friend int BookMenu*( )

Purpose:

This declaration specifies that BookMenu( ), which is a menu function for Book operations, is a friend of this class.

7. *friend Book operator* = (Book B1, Book B2)

Purpose:

This declaration specifies that the overloaded = operator function is a friend of this class.

8. *~Books*( )

Purpose:

This member function is a destructor and is used to deallocate space occupied

}

4.2.16 Class Borrowablebooks : Public Books

{

Private:

date Date_borrowed.

int Borrower_nr;

int Card_nr;

int Copy_nr;

date renewal_date;

date return_date;

float Fine_for_the_book;

Public:

1. *void Borrowablebooks*( )

Purpose:

This member function is a constructor and is used to assign initial values.

2.a. *void Showbook*(int book_no)

b. *void Showbook*(int Acc_no)

c. *void Showbook*(char *title)

d. *void Showbook*(char *author)

Purpose:

The above member functions are overloaded functions and are used to retrieve Borrowable book information.

3.a. *void Modifybook*(int book_no)

b. *void Modifybook*(int Acc_no)

c. *void Modifybook*(char *title)

d. *void Modifybook*(char *author)

Purpose:

The above member functions are overloaded functions and are used to modify Borrowable book information.

4. *void Addbook*( )

Purpose:

The above member function is used to add new borrowable book(s).

5. *void Deletebook*(int Book_no)

Purpose:

The above member function is used to delete a borrowable bo ok

6. *friend int BorrBookMenu*( )

Purpose:

This declaration specifies that BorrBookMenu( ), which is a menu function for Borrowable Book operations, is a friend of this c ass

7. *friend class ServiceManager*

Purpose:

The above declaration specifies that Borrowable Books class is a friend of Service Manager class.

8. *friend inline float fine*(int No_of_days, float fine_per_day)

Purpose:

The above declaration defines that fine( ) function, which i s an inline function used to calculate the fine for a book, is a friend of this class.

9. *void diskinborrbook*( )

Purpose:

The above member function is used to write borrowable book s on to the disk.

10. *Borrowablebooks diskoutborrbook*( )

Purpose:

The above member function is used to retrieve borrowable book information from the disk.

11. *~Borrowablebooks*( )

Purpose:

This member function is a destructor and is used to reclaim space.

12. *friend class TransactionManager*

Purpose:

This declaration specifies that Borrowable Books class is a friend of Transaction Manager class.

File associated with this class is Borrowable.dat .

}

4.2.17 Class ReferenceBooks : Public Books

{

Private:

int hours_allotted;

Public:

1. *void Referencebooks*( )

Purpose:

This member function is a constructor and is used to assign initial values.

2.a. *void Showbook*(int book_no)

b. *void Showbook*(int Acc_no)

c. *void Showbook*(char *title)

d. void Showbook(char *author)

Purpose:

The above member functions are overloaded functions and are used to retrieve Reference book information.

3.a *void Modifybook*(int book_no)

b. *void Modifybook*(int Acc_no)

c. *void Modifybook*(char *title)

d. *void Modifybook*(char *author)

Purpose:

The above member functions are overloaded functions and are used to modify Reference book information.

4. *void Addbook*( )

Purpose:

The above member function is used to add new reference book(s).

5. *void Deletebook*(int Book_no)

Purpose:

The above member function is used to delete a reference book.

6. *friend int RefBookMenu*( )

Purpose:

This declaration specifies that RefBookMenu( ), which is a menu function for Reference Book operations, is a friend of this class.

7. *friend class ServiceManager*

Purpose:

The above declaration specifies that Reference class is a friend of Service Manager class.

8. *void diskinrefbook*( )

Purpose:

The above member function is used to write reference books information on to the disk.

9. *Referencebooks diskoutrefbook*( )

Purpose:

The above member function is used to retrieve reference book information from the disk

10. *Referencebooks*( )

Purpose.

This member function is a destructor and is used to reclaim space.

File associated with this class is Reference.dat .

}

4.2.18 Class Periodicals : Public Books

{

Private:

char *Periodicity;

int Issue_nr;

date issue_date;

date Subcription_started;

date Subscription_end;

Public:

1. *void Periodicals*( )

Purpose:

This member function is a constructor and is used to assign initial values.

2.a. *void Showbook*(int book_no)

b. *void Showbook*(int Acc_no)

c. *void Showbook*(char *periodicity)

Purpose:

The above member functions are overloaded functions and are used to retrieve Periodicals information.

3.a. *void Modifybook*(int book_no)

b. *void Modifybook*(int Acc_no)

c. *void Modifybook*(char *periodicity)

Purpose:

The above member functions are overloaded functions and are used to modify Periodicals information.

4. *void Addbook*( )

Purpose:

The above member function is used to add new periodical(s).

5. *void Deletebook*(int Book_no)

Purpose:

The above member function is used to delete a periodical.

6. *friend int PerBookMenu*( )

Purpose:

This declaration specifies that PerBookMenu( ), which is a menu function for Periodicals operations, is a friend of this class.

7. *friend class ServiceManager*

Purpose:

The above declaration specifies that Periodicals class is a friend of Service Manager class.

8. *void diskinper*( )

Purpose:

The above member function is used to write periodicals information on to the disk.

9. *Periodicals diskoutper*( )

Purpose:

The above member function is used to read [illegible] periodicals information from the disk.

10. *friend class Backvolumes*

Purpose:

The above declaration specifies that Periodicals class is a friend of the Backvolumes class.

11. *~Periodicals( )*

Purpose:

The above member function is a destructor and is used to reclaim space.

Note:

Some fields in the base class Books will take Null values in the Periodicals class.

File associated with this class is Periodicals.dat .

}

4.2.19 Class Backvolumes : Public Books

{

Private:

int No_of_issues;

date Starting_date;

date ending_date;

char *Binding_type;

char *Periodical_name;

Public:

1. *void Backvolumes*( )

Purpose:

This member function is a constructor and is used to assign initial values.

2.a. *void Showbook*(int book_no)

b. *void Showbook*(int Acc_no)

c. *void Showbook*(date starting_date)

d. *void Showbook*(char *Periodical_name)

Purpose:

The above member functions are overloaded functions and are used to retrieve Backvolumes information.

3.a. *void Modifybook*(int book_no)

b. *void Modifybook*(int Acc_no)

c. *void Modifybook*(date starting_date)

d. *void Modifybook*(char *Periodical_name)

Purpose:

The above member functions are overloaded functions and are used to modify Backvolumes information.

4. *void Addbook*( )

Purpose:

The above member function is used to add new Backvolume(s).

5. *void Deletebook*(int Book_no)

Purpose:

The above member function is used to delete a backvolume.

6. *friend int BackBookMenu*( )

Purpose:

This declaration specifies that BackBookMenu( ), which is a menu function for Backvolumes operations, is a friend of this class.

7. *friend class ServiceManager*

Purpose:

The above declaration specifies that Backvolumes class is a friend of Service Manager class.

8. *void diskinbackbook*( )

Purpose:

The above member function is used to write backvolumes information on to the disk.

9. *Backvolumes diskoutbackbook*( )

Purpose

The above member function is used to retrieve backvolumes information from the disk.

10. *~Backvolumes*( )

Purpose:

This member function is a destructor and is used to deallocate space.

Note:

Some fields of the base class Books will take Null values in the Backvolumes class.

File associated with this class is Backvolume.dat .

}

4.2.20 Class Employee

{

Private:

```
        int   Employee_no;
        char  *Employee_name;
        char  *Qualification;
        char  *Designation;
        int  pay_scale;
        char  *responsibilities;
```

Public:

1. *void Employee*( )

Purpose:

This member function is a constructor and is used to assign initial values.

2.a. *void showemployee*(int emp_no)

b. *void showemployee*(char *emp_name)

c. *void showemployee*(int pay_scale)

Purpose:

These member functions are overloaded functions and are used to retrieve information about an employee or set of employees based upon the argument.

3. *Employee Modifyemployee*(int emp_no)

Purpose:

This member function is used to modify information about and employee. It returns the modified employee object.

4. *Employee *Addemployee*( )

Purpose:

This member function is used to add new employee(s) and return the pointers to newly added employee object(s).

5. *void Deleteemployee*(int emp_no)

Purpose:

This member function is used to delete an employee from the Employee.dat file.

6. *void diskinemployee*( )

Purpose:

This member function is used to write the employee information onto the Employee.dat file for persistence.

7. *Employee diskoutemployee*( )

Purpose:

This member function is used to read the employee information from the disk.

8. *void PrintPaybill*( )

Purpose:

This member function is used to print the pay bill of the employee.

9. *friend int Empmenu( )*

Purpose:

This declaration specifies that the Empmenu( ), which is a menu function for employee transactions, is a friend of this class.

10. *~Employee( )*

Purpose:

This member function is a destructor.

File associated with this class is Employee.dat .

}

4.2.21.Class Income

{

Private:

```
        Int   Month/Year;
        Float Localfunds;
        Float State_contribution;
        Float Central_contribution;
        Float Interest_on_endowments;
        Float Fines_and_fees;
        Struct Other
                    {
                     Float Xerox_revenue;
                     Float Binding_revenue;
                     Float etc;
                    };
```

Public:

1. *void Income( )*

Purpose:

This member function is a constructor and is used to assign initial values

2. *void ShowIncome*(Int Month/year)

Purpose:

This member function is used to retrieve Income information for the particular month of a year.

3. *Income ModifyIncome*(Int Month/year)

Purpose:

This member function is used to modify income information for the particular month in a year. It returns the newly modified income object.

4. *Income Addincome*( )

Purpose:

This member function is used to add new income object, and return the newly added Income object.

formation on to

me information

tion, which is a

of this class

5. *void diskinincome*(

Purpose:

This member function is used to write Income in

the Income.dat file for persistence.

6. *Income diskoutincome*( )

Purpose:

This member function is used to retrieve Inco

from the Income.dat file.

7. *friend int IncMenu*( )

Purpose:

This declaration specifies that the IncMenu( ) func

menu function for Income operations, is a friend function

8. *~Income*( )

Purpose:

This member function is a destructor.

File associated with this class is Income.dat .

}

4.2.22 Class Expenditure

{

Private:

int   Month/year; (mm/yy)

Float Salary_and_wages;

Float Amount_on_books;

Float Amount_on_periodicals;

Float Amount_on_newspapers;

Float Electricity_charges;

Float Maintenance_charges;

Float Miscellaneous;

Public:

1. *void Expenditure*( )

Purpose:

This member function is a constructor and is used to assign initial values.

2. *void Showexpenditure*(int month/year)

Purpose:

This member function is used display the expenditure information given the particular month in a year.

3. *Expenditure ModifyExpenditure*(int month/year)

Purpose:

This member function is used to modify the information about the expenditure of a particular month in a year. It returns the modified expenditure object.

4. *Expenditure AddExpenditure*( )

Purpose:

This member function is used to add new expenditure information, and returns the newly added expenditure object.

5. *void diskinexp*( )

Purpose:

This member function is used to write information on to the Expenditure.dat file for persistence.

6. *Expenditure diskoutexp*( )

Purpose:

This member function is used to read information from the Expenditure.dat file.

7. *friend int ExpMenu*( )

Purpose:

This declaration specifies that the ExpMenu( ), which is a function that presents a menu to the user to choose any of the different options in expenditure transactions, is a friend function of this class.

8. *~Expenditure*( )

Purpose:

This member fucntion is a destructor.

File associated with this class is Expenditure.dat .

}

4.2.23 Class Finance

{

Private:

Income inc;

Expenditure exp;

Public:

1. *friend int finmenu*( )

Purpose:

This declaration specifies that the finmenu( ), which is a menu for Finance operations, is a friend of this class.

2. *void showfinance*(int month/year)

Purpose:

This member function used to display financial information calls the show functions of Income and Expenditure classes.

3. *void Modifyfinance*(int month/year)

Purpose:

This member function used to modify financial information calls the modify functions of Income and Expenditure classes.

4. *void Addfinance*( )

Purpose:

This member function used to add new financial information calls the add functions in Income and Expenditure classes.

5. *void diskinfinance*( )

Purpose:

This member function calls the diskin function of Income and Expenditure classes to store information on the disk.

6. *void diskoutfinance*( )

Purpose:

This member function calls the diskout functions of Income and Expenditure classes to read information from the disk.

1

4.2.24 Class MiscellaneousManager

{

private:

Employee emp;

Finance fin;

public:

1. *friend int Miscmenu*( )

Purpose:

This declaration specifies that the Miscmenu( ), which is a menu function for the user to choose from different options based upon his transaction (could be finance or employee based transactions), is a friend function of this class.

2. *void Showmisc*(int choice)

Purpose:

This member function used to display employee information or finance information or both depending upon the choice given, calls the show function(s) in the Employee and/or Finance classes.The choice is obtained from the Miscmenu( ) function.

3. *void Modifymisc*(int choice)

Purpose:

This member function used to modify employee or finance information or both depending upon the choice given, calls the modify function(s) in the Employee and/or Finance classes. The choice is obtained from the Miscmenu( ) function.

4. *void diskinmisc*( )

Purpose:

This member function calls the diskin member functions of [illegible]

Employee and Finance classes.

5. *void diskoutmisc*( )

Purpose:

This member function calls the diskout member functions of Employee and Finance classes.

6. *friend class LibraryManager.*

Purpose:

This declaration specifies that Miscellaneous Manager class is friend of Library Manager class.

}

The above descriptions give the member data and functions of each class given in the design. Other than these there are a few utilities that are used to maintain the flow of information in the software developed.

Chapter 5

CONCLUSIONS

5.1 Observations

An Object Oriented Approach to the Library Information System gives a higher level of abstraction that appeals to the workings of the human cognition.

The Inherent Complexity involved in a big system like Library can be easily usurped using an Object Model. As said earlier not only the software but also the entire OOD can be used thus increasing reusability.

The Object Model is more resilient to changes which means that this system can evolve over a period of time, rather than be abandoned or completely redesigned in response to the first change in the requirements.

Furthermore the Object Model is more smaller than its Non-Object-Oriented implementations. Not only does this mean less code to write and maintain but also translates into cost and schedule benefits.

Finally as Robson [8] says " Many people who have no idea of how a Computer works find the idea of Object Oriented Systems quite natural".

5.2 Limitations

No performance analysis is made about the read and write operations of the file streams in Borland C++.

A few utilities and Indexing of records could not be done due to severe time constraints.

5.3 Future

Extensions be easily made as the entire design is readily available. For example, a person may want to add Binding operations of the library to the existing system. He can do so by creating a Class called Binding and accommodate it in to the design available with minimum fuss.

Classes may be removed, coerced or split as the specifications and requirements for different libraries change.

Windows interface can be given, as the implementation is in Borland C++, thus making the software user friendly.

Indexing of records can be done to enhance the software performance.

Appendix A

BIBLIOGRAPHY

[1].*A.K.Mukherji*, **"Book selection & systematic bibliography "**, World press, 1968.

[2]. *B.Guha*, **"Journal of Library & Information Sciences"**, PP 108-117, June 1976.

[3]. *Bjarne Stroustrup*, **"The C++ Programming Language"** I Edition, Addison Wesley, 1986.

[4]. *Brad.J.Cox*, **"Object Oriented programming , An evolutionary approach"**, I edition, Addition-Wesley, 1987.

[5]. *Bruce Eckel*, **"C++ Inside Out"**, McGraw Hill.

[6]. *David E.Monarchi*, **"A Research Typology for Oriented Analysis & Design"**, CACM[35-47], September 1992.

[7]. *David L.Clark*, **"Database Design"**, McGraw Hill, 1991.

[8]. *D.Robson*, **"Object-Oriented Software System"**, Byte, Vol6(8), August 1991.

[9]. *Edward V.Berard*, **"Essays in Object-Oriented Software Engg"**, I Edition, PHI, 1993.

[10]. *Grady Booch*, **"Object-Oriented Design with Applications"**, Benjamin/Cummings Publishing Company, 1992.

[11]. *Grady Booch*, **" Object-Oriented Analysis & Design With Applications"**, II Edition, Benjamin/Cummings Publishing Company, 1993.

[12].*Greenfell.David*, **"Periodicals and serials "**, London, ASLIB, 1973, p114.

[13]. *Hoare.C*, **" Program: Sorcery or Science"**, Software, April 84,

Vol1(2), IEEE.

[14]. *James Rumbaugh et.al*, **"Object-Oriented Modelling & Design"**, I edition, PHI, 1991.

[15]. *Liskov.B*, **"Data Abstraction & Hierarchy"**, SIGPLAN Notices Vol.23(5), May 1988.

[16]. ***P.S.G.Kumar***, **"Computerization in Indian Libraries"** , B. R. publishing Corporation, 1987.

[17]. *R.L.Mittal*, "Library Administration Theory & Practice", Metropolitan Book Company(Pvt) Limited, 1987.

[18]. *Robert Frichman*, "Object Oriented & Conventional Analysis & Design Methodologies", COMPUTER[22-39], October 1992, IEEE.

[19]. *Robert Lafore*, **"Turbo C++ "**, Waite Group, 1994.

[20]. ***S.P.Singh***, **"Automation in Indian Libraries "**, Metropolitan, New Delhi, 1975.

[21]. *Senn A.James*, **"Analysis and Design of Information Systems"**, II Edition, McGraw Hill, 1989.

[22]. ***S.R.Ranganathan & Gopinath***," **Library Book Acquisition & Cataloguing"**, Asia Publishing House, 1966.

[23]. ***Stanley B.Lippman***, **"C++ Primer"**, I Edition, Addison Wesley Publishing Company, 1991.

[24]. ***William Bishop Warner*** , **"Hand book of Modern library Cataloguing"**, Baltimore, williams and wilkins, 1924.