- Description/purpose
- Related data items
- Range of values
- Data structure definition/form

The name of the data item is self-explanatory. Aliases include other names by which this data item is called *e.g.*, DEO for Data Entry Operator and DR for deputy Registrar. Description/Purpose is a textual description of what the data item is used for or why it exists. Related data items capture relationships between data items *e.g.*, total_marks must always equal to internal_marks plus external_marks.

Range of values records all possible values, *e.g.*, total marks must be positive and between 0 to 100. Data flows capture the names of the processes that generate or receive the data item. If data item is primitive, then data structure definition/form captures the physical structure of the data item. If the data is itself a data aggregate, then data structure definition/form captures the composition of the data items in terms of other data items [DAV190]. The mathematical operators used within the data dictionary are defined in Table 3.3 [DEMA79].

Table 3.3: Data dictionary notation and mathematical operators

Notation	*Meaning*
$x = a + b$	x consists of data elements a and b
$x = [a/b]$	x consists of either data element a or b
$x = a$	x consists of an optional data element a
$x = y\{a\}$	x consists of y or more occurrences of data element a
$x = \{a\}z$	x consists of z or fewer occurrences of data element a
$x = y\{a\}z$	x consists of some occurrences of data element a which are between y and z.

The data dictionary can be used to:

- Create an ordered listing of all data items.
- Create an ordered listing of a subset of data items.
- Find a data item name from a description.
- Design the software and test cases.

3.5.3 Entity-Relationship Diagrams

Another tool for requirement analysis is the entity-relationship diagram, often called as "E-R diagram" [CHEN76]. It is a detailed logical representation of the data for an organisation and uses three main constructs *i.e.*, data entities, relationships, and their associated attributes.

Entities

An entity is a fundamental thing of an organisation about which data may be maintained. A entity has its own identity, which distinguishes it from each other entity. An entity type is th description of all entities to which a common definition and common relationships and a tributes apply.

Consider a university that offers both regular and distance education programmes. The programmes are offered to national and international students.

PROGRAMME and STUDENT are both entity types in this example. Regular and di tance education are entities of PROGRAMME whereas national and international are entitie of STUDENT.

We use capital letters in naming an entity type and in an ER diagram the name i placed inside a rectangle representing that entity as shown in Fig. 3.9.

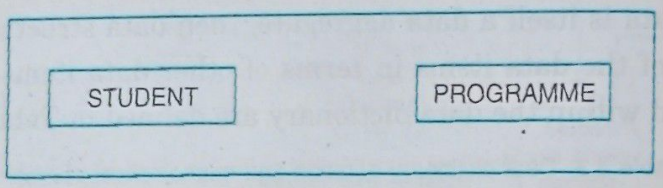


Fig. 3.9: Two entity types in an E-R diagram

Relationships

A relationship is a reason for associating two entity types. These relationships are sometimes called binary relationships because they involve two entity types. Some forms of data model allow more than two entity types to be associated. A STUDENT is registered for a PRO-GRAMME. Relationships are represented by diamond notation in the E-R diagram as shown in Fig. 3.10.

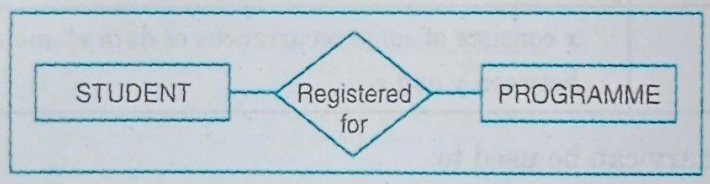


Fig. 3.10: Relationships added to ERD

We consider another example in which, a teaching department of a university in interested in tracking which subjects each of its students has completed. This leads to a relationship called "completes" between the STUDENT and SUBJECT entity types. This is shown in Fig. 3.11.

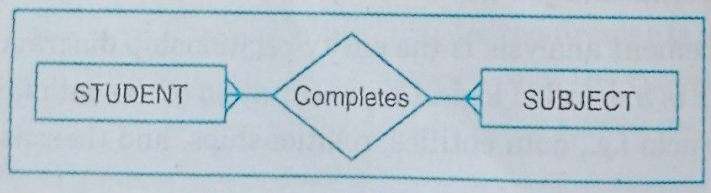


Fig. 3.11: Relationships in ERD

As indicated by the arrows, this is a many-to-many relationship. Each student may complete more than one subject, and more than one student may complete each subject.

Degree of relationships

The degree of a relationship is the number of entity types that participate in that relationship. Thus, relationship "completes" shown in Fig. 3.11 is of degree two, since there are two entity types: STUDENT and SUBJECT. The three most common relationships in E-R models are unary (degree 1), binary (degree 2), and ternary (degree three). Higher-degree relationships are possible, but they are rarely encountered in practice.

Unary relationship

This is also called recursive relationship. It is a relationship between the instances of one entity type. An instance is a single occurrence of an entity type. There may be many instances of an entity type. For example, there is one STUDENT entity type in universities, but there may be hundreds of instances of this entity type in the database. In Fig. 3.12, Is-Married-to is shown as one to one relationship between instances of the PERSON entity type. That is, each person may be currently married to one other person. In the second example, "Is friend of" is shown as one to many relationships between instances of the STUDENT entity type.

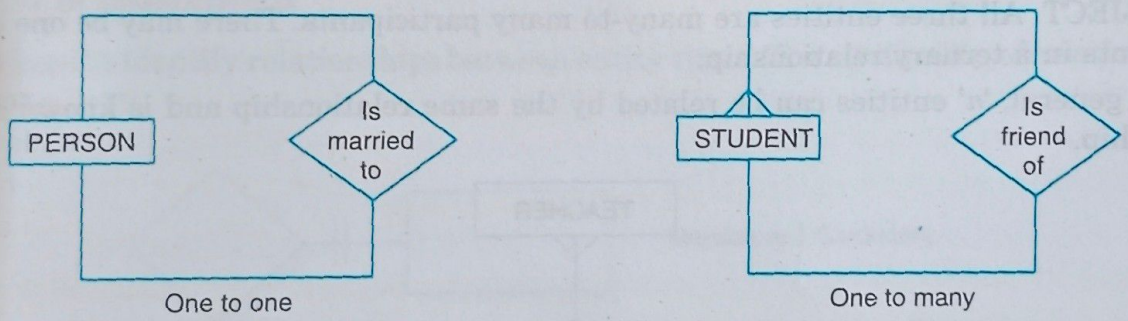


Fig. 3.12: Unary relationships

Binary relationship

It is a relationship between instances of two entity types and is the most common type of relationship encountered in E-R diagrams. Fig. 3.13 shows three examples.

The first (one to one) indicates that a STUDENT is assigned a STUDENT-ID, and each STUDENT-ID is assigned to a STUDENT. The second (one to many) indicates that a PROGRAMME may have many students, and each STUDENT belongs to only one PROGRAMME.

The third (many to many) shows that a STUDENT may register for more than one SUBJECT, and that each SUBJECT may have many STUDENT registrants.

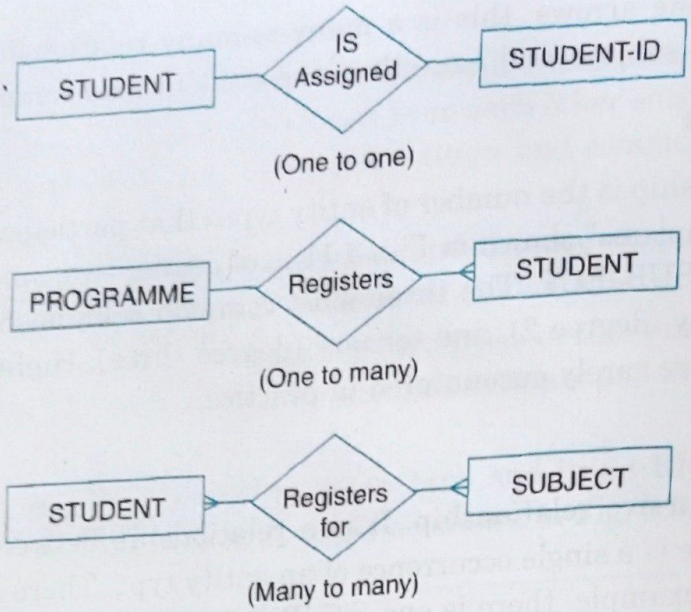


Fig. 3.13: Binary relationships

Ternary relationships

It is a simultaneous relationship amongst instances of three entity types. In Fig. 3.14, the relationship 'may have' provides the association of three entities *i.e.*, TEACHER, STUDENT and SUBJECT. All three entities are many-to many participants. There may be one or many participants in a ternary relationship.

In general, '*n*' entities can be related by the same relationship and is known as *n*-ary relationship.

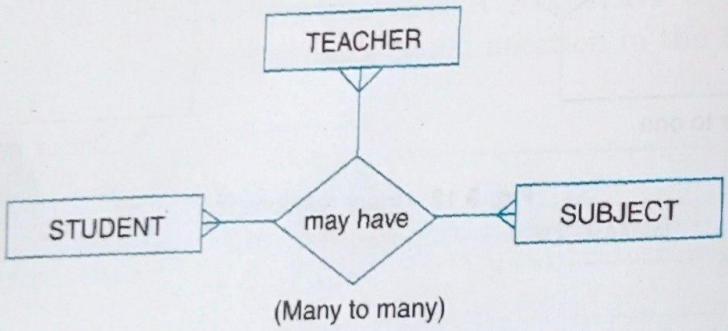


Fig. 3.14: Ternary relationship

Cardinalities and optionality

Suppose that there are two entity types, A and B, that are connected by a relationship. The cardinality of a relationship is the number of instances of entity B that can be associated with each instance of entity A.

Consider the example shown in Fig. 3.15. a student may register for many subjects.

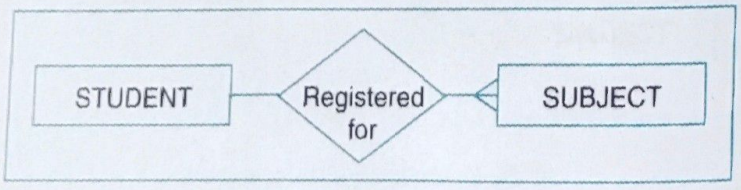


Fig. 3.15: Use of cardinality

In the terminology, we have discussed so far, this example has "one-to many" relationship. Yet it may also be true that a subject may not have any student at specific instance of time. We need a more precise notation to indicate the range of cardinalities for a relationship.

The minimum cardinality of a relationship is the minimum number of instances of entity B that may be associated with each instance of entity A. If minimum number of students available for a subject is zero, we say that subject is an optional participant in the 'register for' relationship. When the minimum cardinality of a relationship is one, then we say entity B is a mandatory participant in the relationship. The maximum cardinality is the maximum number of instances. In our example, maximum is 'many'. The modified E-R diagram is given in Fig. 3.16. The zero through the line near the SUBJECT entity means a minimum cardinality of zero, while the crow's foot notation means a 'many' maximum cardinality.

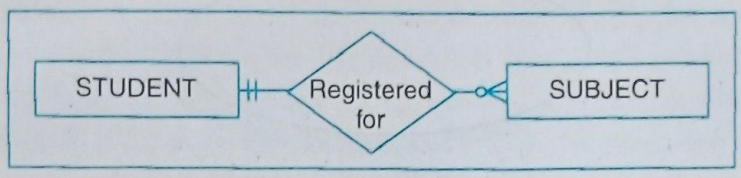


Fig. 3.16: Modified ER diagram

Cardinality of relationships

It can be used to identify relationships between entity types. The cardinality of relationships is given in Fig. 3.17.

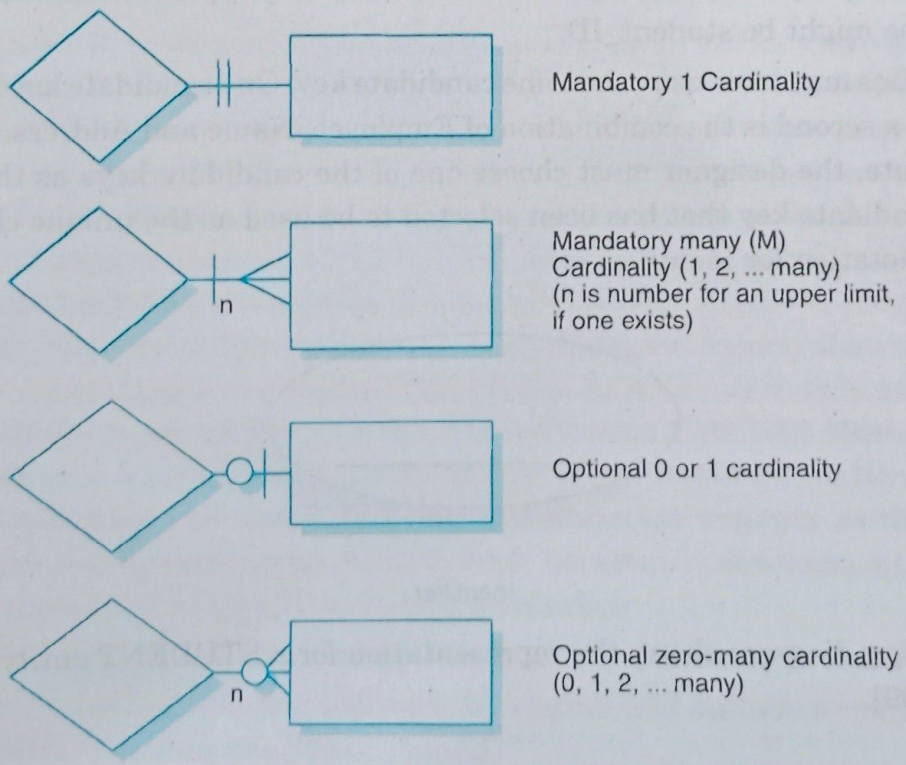


Fig. 3.17: Relationship cardinality [HOFF99]

Attributes

Each entity type has a set of attributes associated with it. An attribute is a property or characteristic of an entity that is of interest to the organization. Following are some typical entity types and associated attributes:

STUDENT: Student_ID, Student_Name, Address, Phone_Number

EMPLOYEE: Employee_ID, Employee_Name, Address.

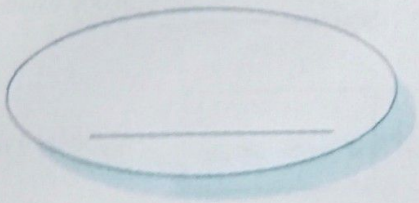
We use an initial capital letter, followed by lowercase letters, and nouns in naming an attribute. In E-R diagram, we can visually represent an attribute by placing its name as an ellipse with a line connecting it to the associated entity. Notation for attribute is

Attribute

Candidate keys and identifier

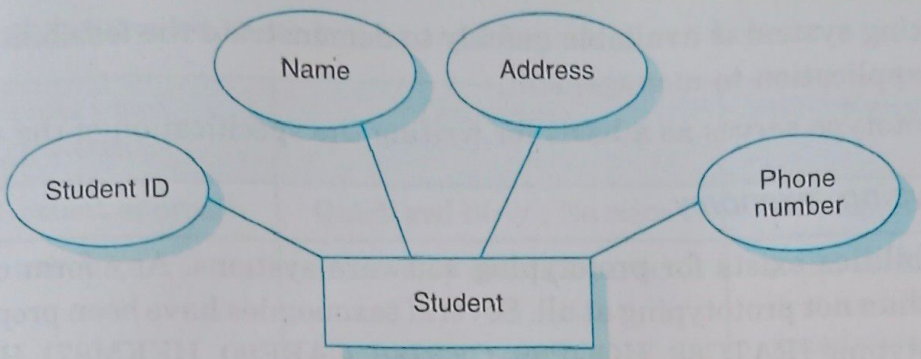
Every entity type must have an attribute or set of attributes that distinguishes one instance from other instances of the same type. A candidate key is an attribute (or combination of attributes) that uniquely identifies each instance of an entity type. A candidate key for a STUDENT entity type might be student_ID.

Some entities may have more than one candidate key. One candidate key for EMPLOYEE is Employee_ID, a second is the combination of Employee_Name and Address. If there is more than one candidate, the designer must choose one of the candidate keys as the identifier. An identifier is a candidate key that has been selected to be used as the unique characteristic for an entity type. Notation for identifier is



Identifier

The following diagram shows the representation for a STUDENT entity type using E-R notation [HOFF99].



Using entity relationship diagram to represent data is still an important technique today. It should not, however, be used in isolation, but together with techniques that fully represent the business objects we encounter every day.

The data flow diagram and the E-R diagram, each highlight a different aspect of the same system. As a consequence, there are one-to-one correspondences that must be checked to ensure that an E-R diagram and a data flow diagram are consistent over all applications. This suggests that it is desirable to use both methods such that we can view the logical issues from the two perspectives generated by these approaches.

3.5.4 Software Prototyping

Prototyping is the technique of constructing a partial implementation of a system so that customers, users, or developers can learn more about a problem or a solution to that problem. It is a partial implementation because if it were full implementation, it would be the system, not a prototype of it.

It allows users to explore and criticize proposed systems before undergoing the cost of a full-scale development. The field of prototyping software systems has emerged around two prototyping technologies, *i.e.*, throw-away and evolutionary. In throw-away approach, the prototype software is constructed in order to learn about the problem or its solution and is usually discarded after the desired knowledge is gained. In the evolutionary approach, the prototype is constructed in order to learn about the problem or its solution in successive steps. Once the prototype has been used and the requisite knowledge is gained, the prototype is then adapted to satisfy the, now better-understood, needs. The prototype is then used again, more is learned, and the prototype is re-adapted. This process repeats indefinitely until the prototype system satisfies all needs and thus evolves into the real system [DAV190]. Hence, in evolutionary prototyping the focus is on achieving functionality for demonstrating a portion of the system to the end user for feedback and system growth. The prototype emerges as the actual system downstream in the software life cycle. As with each iteration in development, functionality is added and then translated to an efficient implementation.

The benefits of developing a prototype early in the software process are [SOMM96]

1. Misunderstanding between software developers and customers may be identified as the system functions are demonstrated.
2. Missing user requirements may be detected.
3. Difficult-to-use or confusing user requirements may be identified and refined.

94613 LIBRARY VAZHUTHALA

shall never enter an infinite loop" is non-verifiable because the testing of this quality is theoretically impossible.

An example of a verifiable statement is "output of the program shall be produced within 20 seconds of event × 60% of the time; and shall be produced within 30 seconds of event × 100% of the time." This statement can be verified because it uses concrete terms and measurable quantities.

If a method cannot be devised to determine whether the software meets a particular requirement, then that requirement should be removed or revised.

Modifiable

The SRS is modifiable if, and only if, its structure and style are such that any changes to the requirements can be made easily, completely, and consistently while retaining the structure and style.

The requirements should not be redundant. Redundancy itself is not an error, but it can easily lead to errors. Redundancy can occasionally help to make an SRS more readable, but a problem can arise when the redundant document is updated. For instance, a requirement may be altered in only one of the places out of the many places where it appears.

The SRS then becomes inconsistent. Whenever redundancy is necessary, the SRS should include explicit cross-references to make it modifiable.

Traceable

The SRS is traceable if the origin of each of the requirements is clear and if it facilitates the referencing of each requirement in future development or enhancement documentation. Two types of traceability are recommended.

1. Backward traceability: This depends upon each requirement explicitly referencing its source in earlier documents.
2. Forward traceability: This depends upon each requirement in the SRS having a unique name or reference number.

The forward traceability of the SRS is especially important when the software product enters the operation and maintenance phase. As code and design documents are modified, it is essential to be able to ascertain the complete set of requirements that may be affected by those modifications.

3.6.3 Organization of the SRS

The Institute of Electrical and Electronics Engineers (IEEE) has published guidelines and standards to organize an SRS document [IEEE87, IEEE94]. Different projects may require their requirements to be organized differently, that is, there is no one method that is suitable for all projects. It provides different ways of structuring the SRS. The first two sections of the SRS are the same in all of them. The specific tailoring occurs in section 3 entitled "specific requirements". The general organisation of an SRS is given in Fig. 3.18 [IEEE93].

1. Introduction

The following subsections of the Software Requirements Specifications (SRS) document should provide an overview of the entire SRS.

1.1 Purpose

Identify the purpose of this SRS and its intended audience. In this subsection, describe the purpose of the particular SRS and specify the intended audience for the SRS.

1.2 Scope

In this subsection:

(*i*) Identify the software product(s) to be produced by name

(*ii*) Explain what the software product(s) will, and, if necessary, will not do

(*iii*) Describe the application of the software being specified, including relevant benefits, objectives, and goals

(*iv*) Be consistent with similar statements in higher-level specifications if they exist.

1.3 Definitions, Acronyms, and Abbreviations

Provide the definitions of all terms, acronyms, and abbreviations required to properly interpret the SRS. This information may be provided by reference to one or more appendices in the SRS or by reference to documents. This information may be provided by reference to an Appendix.

1.4 References

In this subsection:

(*i*) Provide a complete list of all documents referenced elsewhere in the SRS

(*ii*) Identify each document by title, report number (if applicable), date, and publishing organisation

(*iii*) Specify the sources from which the references can be obtained.

This information can be provided by reference to an appendix or to another document.

1.5 Overview

In this subsection:

(*i*) Describe what the rest of the SRS contains

(*ii*) Explain how the SRS is organised.

2. The Overall Description

Describe the general factors that affect the product and its requirements. This section does not state specific requirements. Instead, it provides a background for those requirements, which are defined in section 3, and makes them easier to understand.

2.1 Product Perspective

Put the product into perspective with other related products. If the product is independent and totally self-contained, it should be so stated here. If the SRS defines a product that is a component of a larger system, as frequently occurs, then this subsection relates the require-

ments of the larger system to functionality of the software and identifies interfaces between that system and the software.

A block diagram showing the major components of the large system, interconnections, and external interfaces can be helpful.

The following subsections describe how the software operates inside various constraints.

2.1.1 System Interfaces

List each system interface and identify the functionality of the software to accomplish the system requirement and the interface description to match the system.

2.1.2 Interfaces

Specify:

(*i*) The logical characteristics of each interface between the software product and its users.

(*ii*) All the aspects of optimizing the interface with the person who must use the system.

2.1.3 Hardware Interfaces

Specify the logical characteristics of each interface between the software product and the hardware components of the system. This includes configuration characteristics. It also covers such matters as what devices are to be supported, how they are to be supported and protocols.

2.1.4 Software Interfaces

Specify the use of other required software products and interfaces with other application systems. For each required software product, include:

(*i*) Name

(*ii*) Mnemonic

(*iii*) Specification number

(*iv*) Version number

(*v*) Source

For each interface, provide:

(*i*) Discussion of the purpose of the interfacing software as related to this software product

(*ii*) Definition of the interface in terms of message content and format.

2.1.5 Communications Interfaces

Specify the various interfaces to communications such as local network protocols, etc.

2.1.6 Memory Constraints

Specify any applicable characteristics and limits on primary and secondary memory.

2.1.7 Operations

Specify the normal and special operations required by the user such as:

(*i*) The various modes of operations in the user organisation

(*ii*) Periods of interactive operations and periods of unattended operations

(*iii*) Data processing support functions

(*iv*) Backup and recovery operations.

2.1.8 Site Adaptation Requirements

In this section:

(*i*) Define the requirements for any data or initialization sequences that are specific to a given site, mission, or operational mode

(*ii*) Specify the site or mission-related features that should be modified to adapt the software to a particular installation.

2.2 Product Functions

Provide a summary of the major functions that the software will perform. Sometimes the function summary that is necessary for this part can be taken directly from the section of the higher-level specification (if one exists) that allocates particular functions to the software product.

For clarity:

(*i*) The functions should be organised in a way that makes the list of functions understandable to the customer or to anyone else reading the document for the first time.

(*ii*) Textual or graphic methods can be used to show the different functions and their relationships. Such a diagram is not intended to show a design of a product but simply shows the logical relationships among variables.

2.3 User Characteristics

Describe those general characteristics of the intended users of the product including educational level, experience, and technical expertise. Do not state specific requirements but rather provide the reasons why certain specific requirements are later specified in sections 3.

2.4 Constraints

Provide a general description of any other items that will limit the developer's options. These can include:

(*i*) Regulatory policies

(*ii*) Hardware limitations (for example, signal timing requirements)

(*iii*) Interface to other applications

(*iv*) Parallel operation

(*v*) Audit functions

(*vi*) Control functions

(*vii*) Higher-order language requirements

(*viii*) Signal handshake protocols (for example, XON-XOFF, ACK-NACK)

(*ix*) Reliability requirements

(*x*) Criticality of the application

(*xi*) Safety and security considerations.

2.5 Assumptions and Dependencies

List each of the factors that affect the requirements stated in the SRS. These factors are not design constraints on the software but are, rather, any changes to them that can affect the requirements in the SRS. For example, an assumption might be that a specific operating system would be available on the hardware designated for the software product. If, in fact, the operating system were not available, the SRS would then have to change accordingly.

2.6 Apportioning of Requirements

Identify requirements that may be delayed until future versions of the system.

3. Specific Requirements

This section contains all the software requirements at a level of detail sufficient to enable designers to design a system to satisfy those requirements, and testers to test that the system satisfies those requirements. Throughout this section, every stated requirement should be externally perceivable by users, operators, or both external systems. These requirements should include at a minimum a description of every input into the system, every output from the system and all functions performed by the system in response to an input or in support of an output. The following principles apply:

(*i*) Specific requirements should be stated with all the characteristics of a good SRS
- correct
- unambiguous
- complete
- consistent
- ranked for importance and/or stability
- verifiable
- modifiable
- traceable

(*ii*) Specific requirements should be cross-referenced to earlier documents that relate

(*iii*) All requirements should be uniquely identifiable

(*iv*) Careful attention should be given to organizing the requirements to maximize readability.

Before examining specific ways of organising the requirements it is helpful to understand the various items that comprise requirements as described in the following sebsections.

3.1 External Interfaces

This contains a detailed description of all inputs into and outputs from the software system. It complements the interface descriptions in *section 2 but does* not repeat information there. It contains both content and format as follows:

- Name of item
- Description of purpose
- Source of input or destination of output
- Valid range, accuracy and/or tolerance
- Units of measure

- Timing
- Relationships to other inputs/outputs
- Screen formats/organisation
- Window formats/organisation
- Data formats
- Command formats
- End messages.

3.2 Functions

Functional requirements define the fundamental actions that must take place in the software in accepting and processing the inputs and in processing and generating the outputs. These are generally listed as "shall" statements starting with "The system shall...

These include:

- Validity checks on the inputs
- Exact sequence of operations
- Responses to abnormal situation, including
  - Overflow
  - Communication facilities
  - Error handling and recovery
- Effect of parameters
- Relationship of outputs to inputs, including
  - Input/Output sequences
  - Formulas for input to output conversion.

It may be appropriate to partition the functional requirements into sub-functions or sub-processes. This does not imply that the software design will also be partitioned that way.

3.3 Performance Requirements

This subsection specifies both the static and the dynamic numerical requirements placed on the software or on human interaction with the software, as a whole. Static numerical requirements may include:

(*i*) The number of terminals to be supported

(*ii*) The number of simultaneous users to be supported

(*iii*) Amount and type of information to be handled

Static numerical requirements are sometimes identified under a separate section entitled capacity.

Dynamic numerical requirements may include, for example, the number of transactions and tasks and the amount of data to be processed within certain time periods for both normal and peak workload conditions.

All of these requirements should be stated in measurable terms:

For example,

95% of the transactions shall be processed in less than 1 second rather than,

An operator shall not have to wait for the transaction to complete.

(**Note:** Numerical limits applied to one specific function are normally specified as part of the processing subparagraph description of that function).

3.4 Logical Database Requirements

This section specifies the logical requirements for any information that is to be placed into a database. This may include:

- Types of information used by various functions
- Frequency of use
- Accessing capabilities
- Data entities and their relationships
- Integrity constraints
- Data retention requirements.

3.5 Design Constraints

Specify design constraints that can be imposed by other standards, hardware limitations, etc.

3.5.1 Standards Compliance

Specify the requirements derived from existing standards or regulations. They might include:

(*i*) Report format
(*ii*) Data naming
(*iii*) Accounting procedures
(*iv*) Audit Tracing

For example, this could specify the requirement for software to processing activity. Such traces are needed for some applications to meet minimum regulatory or financial standard. An audit trace requirement may, for example, state that all changes to a payroll database must be recorded in a trace file with before and after values.

3.6 Software System Attributes

There are a number of quality attributes of software that can serve as requirements. It is important that required attributes be specified so that their achievement can be objectively verified. Fig. 3.19 has the definitions of the quality attributes of the software discussed in this subsection [ROBE02]. The following items provide a partial list of examples.

3.6.1 Reliability

Specify the factors required to establish the required reliability of the software system at time of delivery.

3.6.2 Availability

Specify the factors required to guarantee a defined availability level for the entire system such as checkpoint, recovery, and restart.

3.6.3 Security

Specify the factors that would protect the software from accidental or malicious access, use, modification, destruction, or disclosure. Specific requirements in this area could include the need to:

- Utilize certain cryptographic techniques
- Keep specific log or history data sets
- Assign certain functions to different modules
- Restrict communications between some areas of the program
- Check data integrity for critical variables.

3.6.4 Maintainability

Specify attributes of software that relate to the ease of maintenance of the software itself. There may be some requirement for certain modularity, interfaces, complexity, etc. Requirements should not be placed here just because they are thought to be good design practices.

3.6.5 Portability

Specify attributes of software that relate to the ease of parting the software to other host machines and/or operating systems. This may include:

- Percentage of components with host-dependent code
- Percentage of code that is host dependent
- Use of a proven portable language
- Use of a particular compiler or language subset
- Use of a particular operating system.

S. No.	Quality Attributes	Definition
1.	Correctness	extent to which program satisfies specifications, fulfills user's mission objectives
2.	Efficiency	amount of computing resources and code required to perform function
3.	Flexibility	effort needed to modify operational program
4.	Interoperability	effort needed to couple one system with another
5.	Reliability	extent to which program performs with required precision
6.	Reusability	extent to which it can be reused in another application
7.	Testability	effort needed to test to ensure performance as intended
8.	Usability	effort required to learn, operate, prepare input, and interpret output
9.	Maintainability	effort required to locate and fix an error during operation
10.	Portability	effort needed to transfer from one hardware or software environment to another.
11.	Integrity/security	extent to which access to software or data by unauthorised people can be controlled.

Fig. 3.19: Definitions of quality attributes

3.7 Organising the Specific Requirements

For anything but trivial systems the detailed requirements tend to be extensive. For this reason, it is recommended that careful consideration be given to organising these in a manner optimal for understanding. There is no one optimal organisation for all systems. Different classes of systems lend themselves to different organisations of requirements. Some of these organisations are described in the following subclasses.

3.7.1 System mode

Some systems behave quite differently depending on the mode of operation. When organising by mode there are two possible outlines. The choice depends on whether interfaces and performance are dependent on mode.

3.7.2 User class

Some systems provide different sets of functions to different classes of users.

3.7.3 Objects

Objects are real-world entities that have a counterpart within the system. Associated with each object is a set of attributes and functions. These functions are also called services, methods, or processes. Note that sets of objects may share attributes and services. These are grouped together as classes.

3.7.4 Feature

A feature is an externally desired service by the system that may require a sequence of inputs to effect the desired result. Each feature is generally described as sequence of stimulus-response pairs.

3.7.5 Stimulus

Some systems can be best organised by describing their functions in terms of stimuli.

3.7.6 Response

Some systems can be best organised by describing their functions in support of the generation of a response.

3.7.7 Functional hierarchy

When none of the above organisational schemes prove helpful, the overall functionality can be organised into a hierarchy of functions organised by either common inputs, common outputs, or common internal data access. Data flow diagrams and data dictionaries can be used to show the relationships between and among the functions and data.

3.8 Additional Comments

Whenever a new SRS is contemplated, more than one of the organisational techniques given in 3.7 may be appropriate. In such cases, organise the specific requirements for multiple hierarchies tailored to the specific needs of the system under specification.

There are many notations, methods, and automated support tools available to aid in the documentation of requirements. For the most part, their usefulness is a function of organisation. For example, when organising by mode, finite state machines or state charts may prove helpful, when organising by object, object-oriented analysis may prove helpful; when organising by feature, stimulus-response sequences may prove helpful; when organising by functional hierarchy, data flow diagrams and data dictionaries may prove helpful.

In any of the outlines below, those sections called "Functional Requirement i" may be described in native language, in pseudocode, in a system definition language, or in four sub-sections titled: Introduction, Inputs, Processing, Outputs.

4. Change Management Process

Identify the change management process to be used to identify, log, evaluate, and update the SRS to reflect changes in project scope and requirements.

5. Document Approval

Identify the approvers of the SRS document. Approver's name, signature, and date should be used.

6. Supporting Information

The supporting information makes the SRS easier to use. It includes:

- Table of Contents
- Index
- Appendices

The appendices are not always considered part of the actual requirements specification and are not always necessary. They may include:

(*a*) Sample I/O formats, descriptions of cost analysis studies, results of user surveys

(*b*) Supporting or background information that can help the readers of the SRS

(*c*) A description of the problems to be solved by the software

(*d*) Special packaging instructions for the code and the media to meet security, export, initial loading, or other requirements.

When appendices are included, the SRS should explicitly state whether or not the appendices are to be considered part of the requirements.

Tables on the following pages provide alternate ways to structure section 3 on the specific requirements.

- Side effects.
- Input/Output Assertions.

- Level 2
  - Local data structures (variable etc.)
  - Timing constraints
  - Exception handling (conditions, responses, events)
  - Any other limitations.
- Level 3
  - Body (structured chart, English pseudo code, decision tables, flow charts, etc.)

5.5 IEEE RECOMMENDED PRACTICE FOR SOFTWARE DESIGN DESCRIPTIONS (IEEE STD 1016–1998)

5.5.1 Scope

This is a recommended practice for describing software designs. This is designed by IEEE (Institution of Electricals and Electronics Engineers) and is known as IEEE Standard (IEEE std. 1016–1998) for software design description (SDD). An SDD is a representation of a software system that is used as a medium for communicating software design information.

5.5.2 References

This standard shall be used in conjunction with the following publications.

(*i*) IEEE std 830–1998, IEEE recommended practice for software requirements specifications

(*ii*) IEEE std 610.12–1990, IEEE glossary of software engineering terminology.

5.5.3 Definitions

Few important definitions are given below:

(*i*) **Design entity:** An element (component) of a design that is structurally and functionally distinct from other elements and that is separately named and referenced.

(*ii*) **Design View:** A subset of design entity attribute information that is specifically suited to the needs of a software project activity.

(*iii*) **Entity attribute:** A named property or characteristics of a design entity. It provides a statement of fact about the entity.

(*iv*) **Software design description (SDD):** A representation of a software system created to facilitate analysis, planning, implementation and decision making. A blue print or model of the software system. The SDD is used as the primary medium for communicating software design information.

5.5.4 Purpose of an SDD

The SDD shows how the software system will be structured to satisfy the requirements identified in the SRS. It is basically the translation of requirements into a description of the software

structure, software components, interfaces, and data necessary for the implementation phase. Hence, SDD becomes the blue print for the implementation activity.

5.5.5 Design Description Information Content

(*i*) **Introduction:** The SDD is a representation or model of the system to be created. The model should provide the precise design information needed for planning and implementation of the software system. It should represent partitioning of the system into design entities and describe the important properties and relationship among those entities.

(*ii*) **Design entities:** A design entity is an element (component) of a design that is structurally and functionally distinct from other elements and that is separately named and referenced.

Design entities result from a decomposition of the software system requirements. The objective is to divide the system into separate components that can be considered, implemented, changed, other components.

The entities may have different nature, but may have common characteristics. Each design entity will have a name, purpose, and function. There are common relationship among entities such as interfaces or shared data. The common characteristics of entities are described by design entity attributes.

(*iii*) **Design entity attributes:** A design entity attribute is a named characteristic or property of a design entity. It provides a statement of fact about the entity.

Design attributes can be thought of as questions about design entities. The answer to those questions are the values of the attributes. All the questions can be answered, but the content of the answer will depend upon the nature of the entity. The collection of answers provides a complete description of an entity.

All attributes shall be specified for each entity. Attribute descriptions should include references and design considerations such as tradeoffs and assumptions when appropriate. In some cases, attribute descriptions may have the value none. When additional attributes are identified for a specific software project, they should be included in the design description. The attributes and associated information items are defined in the following subsections (from '*a*' to '*j*').

(*a*) ***Identification:*** The name of the entity: Two entities shall not have the same name. The names for the entities may be selected to characterise their nature. This will simplify referencing and tracking in addition to providing identification.

(*b*) ***Type:*** A description of the kind of entity. The type attribute shall describe the nature of entity. It may simply name the kind of entity, such as subprogram, module, procedure, process, or data store. Alternatively, design entities may be grouped into major classes to assist in locating an entity dealing with a particular type of information. For a given design description, the chosen entity types shall be applied consistently.

(*c*) ***Purpose:*** A description of why the entity exists. The purpose attribute shall provide the rationale for the creation of the entity. Therefore, it shall designate the specific functional and performance requirements for which the entity is created.

(*d*) ***Function:*** A statement of what the entity does. The function attribute shall state the transformation applied by the entity to inputs to produce the desired output. In the case of

a data entity, this attribute shall state the type of information stored or transmitted by the entity.

(*e*) ***Subordinates:*** The identification of all entities composing this entity. The subordinates attribute shall identify the "composed of relationship" for an entity. This information is used to trace requirements to design entities and to identify parent/child structural relationships through a software system decomposition.

(*f*) ***Dependencies:*** A descriptions of the relationships of this entity with other entities. The dependencies attribute shall identify the uses or requires the presence of relationship for an entity. These relationships are often graphically depicted by structure charts, data flow diagrams etc.

(*g*) ***Interface:*** A description of how other entities interact with this entity. The interface attribute shall describe the methods of interaction and the rules governing those interactions. The methods of interaction include the mechanisms for invoking or interrupting the entity, for communicating through parameters, common data areas or messages, and for direct access to internal data. The rules governing the interaction include the communications protocol, data format, acceptable values, and the meaning of each value.

This attribute shall provide a description of the input ranges, the meaning of inputs and outputs, the type and format of each input or output error codes. For information systems, it should include inputs, screen formats, and a complete description of the interactive language.

(*h*) ***Resources:*** A description of the elements used by the entity that are external to the design. The resources attribute shall identify and describe all of the resources external to the design that are needed by this entity to perform its function. The interaction rules and methods for using the resource shall be specified by this attribute.

This attribute provides information about items such as physical devices (like printers), software services (like math libraries, OS services), and processing resources (like CPU cycles, memory allocation, buffers).

(*i*) ***Processing:*** A description of the rules used by the entity to achieve its function. The processing attribute shall describe the algorithm used by the entity to perform a specific task and shall include contingencies. This description is a refinement of the function attribute.

(*j*) ***Data:*** A description of data elements internal to the entity. The data attribute shall describe the method of representation, initial values, use, semantics, format, and acceptable values of internal data.

The description of data may be in the form of a data dictionary that describes the content, structure, and use of all data elements. Data information shall describe everything pertaining to the use of data or internal data structures by this entity. It shall include data specifications such as formats, number of elements, and initial values. It shall also include the structures to be used for representing data such as file structures, arrays, stacks, queues, and memory partitions.

5.5.6 Design Description Organisation

Each design description writer may have a different view of what are considered the essential aspects of a software design. The organisation of SDD is given in Table 5.1. This is one of the possible ways to organise and format the SDD.

Table 5.1: Organisation of SDD

1. Introduction
   - 1.1 Purpose
   - 1.2 Scope
   - 1.3 Definitions and acronyms
2. References
3. Decomposition description
   - 3.1 Module decomposition
     - 3.1.1 Module 1 description
     - 3.1.2 Module 2 description
   - 3.2 Concurrent process decompostion
     - 3.2.1 Process 1 description
     - 3.2.2 Process 2 description
   - 3.3 Data decomposition
     - 3.3.1 Data entity 1 description
     - 3.3.2 Data entity 2 description
4. Dependency description
   - 4.1 Intermodule dependencies
   - 4.2 Interprocess dependencies
   - 4.3 Data dependencies
5. Interface description
   - 5.1 Module Interface
     - 5.1.1 Module 1 description
     - 5.1.2 Module 2 description
   - 5.2 Process interface
     - 5.2.1 Process 1 description
     - 5.2.2 Process 2 description
6. Detailed design
   - 6.1 Module detailed design
     - 6.1.1 Module 1 detail
     - 6.1.2 Module 2 detail
   - 6.2 Data detailed design
     - 6.2.1 Data entry 1 detail
     - 6.2.2 Data entry 2 detail

Entity attribute information can be organised in several ways to reveal all of the essential aspects of a design. There may be number of ways to view the design. Hence, each design view represents a separate concern about a software system. Together, these views provide a comprehensive description of the design in a concise and usable form that simplifies information access and assimilation. A recommended organisation of the SDD into separate design views to facilitate information access and assimilation is given in Table 5.2.

Table 5.2: Design views

Design view	Scope	Entity attribute	Example representation
Decompostion description	Partition of the system into design entities.	Identification, type purpose, function, subordinate	Hierarchical decomposition diagram, natural language
Dependency description	Description of relationships among entities of system resources	Identification, type, purpose, dependencies, resources	Structure chart, data flow diagrams, transaction diagrams
Interface description	List of everything a designer, developer, tester needs to know to use design entities that make up the system	Identification, function, interfaces	Interface files, parameter tables
Detail description	Description of the internal design details of an entity	Identification, processing, data	Flow charts, PDL etc.

5.6 OBJECT ORIENTED DESIGN

"Object Oriented" has clearly become the buzzword of choice in the industry. Almost everyone talks about it. Almost everyone claims to be doing it, and almost everyone says it is better than traditional function oriented design. Object oriented design is the result of focusing attention not on the function performed by the program, but instead on the data that are to be manipulated by the program. Thus, it is orthogonal to function oriented design.

Object Oriented Design begins with an examination of the real world "things" that are part of the problem to be solved. These things (which we will call objects) are characterized individually in terms of their attributes (transient state information) and behaviour (functional process information). Each object maintains its own state, and offers a set of services to other objects. Shared data areas are eliminated and objects communicate by message passing (e.g. parameters). Objects are independent entities that may readily be changed because all state and representation information is held within the object itself. Objects may be distributed and may execute either sequentially or in parallel [BOOC03].

5.6.1 Basic Concepts

Object Oriented Design is not dependent on any specific implementation language. Problems are modelled using objects. Objects have:

- Behaviour (they do things)
- State (which changes when they do things)

For example, a car is an object. It has state: whether its engine is running; and it has a behaviour: starting the car, which changes its state from "engine not running" to "engine running".

The various terms related to object oriented design are Objects, Classes, Abstraction, Inheritance and Polymorphism.

(*i*) **Objects:** The word "Object" is used very frequently and conveys different meaning in different circumstances. Here, meaning is an entity able to save a state (information) and which offers a number of operations (behaviour) to either examine or affect this state. Hence, an object is characterised by number of operations and a state which remembers the effect of these operations [JACO 98].

All objects have unique identification and are distinguishable. Two bananas may be of same colour, shape and texture but still are different. Each has an identity. There may be four dogs of same colour, breed and size but all are distinguishable. The term identity means that objects are distinguished by their inherent existence and not by descriptive properties [JOSHO 3].

As discussed above, object is an entity, which has a state (whose representation is hidden from other objects) and a defined set of operations, which operate on that state. The state is represented as a set of object attributes. The operations associated with the object provide services to other objects, which request these services when some computation is required. In principle, objects communicate by passing messages to each other and these messages initiate object operations.

(*ii*) **Messages:** Conceptually, objects communicate by message passing. Messages consist of the identity of the target object, the name of the requested operation and any other operation needed to perform the function. In some distributed systems, object communications are implemented directly as text messages which objects exchange. The receiving object parses the message, identifies the service and the associated data and carries out the requested service. Messages are often implemented as procedure or function calls (name = procedure name, information = parameter list).

(*iii*) **Abstraction:** In object oriented design, complexity is managed using abstraction. *Abstraction is the elimination of the irrelevant and the amplification of the essentials.*

We can teach someone to drive any car using an abstraction. We amplify the essentials: we teach about the ignition and steering wheel, and we eliminate the details, such as details of the particular engine in this car or the way fuel is pumped to the engine where a spark ignites it, it explodes pushing down a piston and driving a crankshaft [FIEL01].

Problems usually have levels of abstraction. We can see a car at a high level of abstraction for driving, but mechanics need to work at a lower level of abstraction. They do care about details and need to know about batteries and engines.

However, there are abstractions at the mechanic's level too. The mechanic might test or charge a battery without caring that inside the battery there is a complex chemical reaction going on. A battery designer would care about these details but would not care about, say, the electronics that goes into the car's stereo.

We have seen that we can look at the details such as the battery or stereo design, and they are separated in manageable chunks, and by using abstraction and ignoring the details, we can also look at the whole car as a manageable chunk.

(*iv*) **Class:** In any system, there shall be number of objects. Some of the objects may have common characteristics and we can group the objects according to these characteristics. This type of grouping in known as a class. Hence, a class is a set of objects that share a common structure and a common behaviour.

We may have a class "car" with objects like Indica, Santro, Maruti, Indigo. These objects are related due to some common characteristics to constitute a class named "car". A class may be object defined as [JACO98]:

"A class represents a template for several objects and describes how these objects are structured internally. Objects of the same class have the same definition both for their operations and for their information structures.

In object oriented system, each object belongs to a class. An objects, that belongs to a certain class is called an instance of that class. We often use object and instance as synonyms. Hence, an instance is an object created from a class. The class describes the structure (behaviour and information) of the instance, while the current state of the instance is defined by the operations performed on the instance.

We may define a class "car" and each object that represents a car becomes an instance of this class. In this class "car", Indica, Santro, Maruti, Indigo are instances of this class as shown in Fig. 5.20.

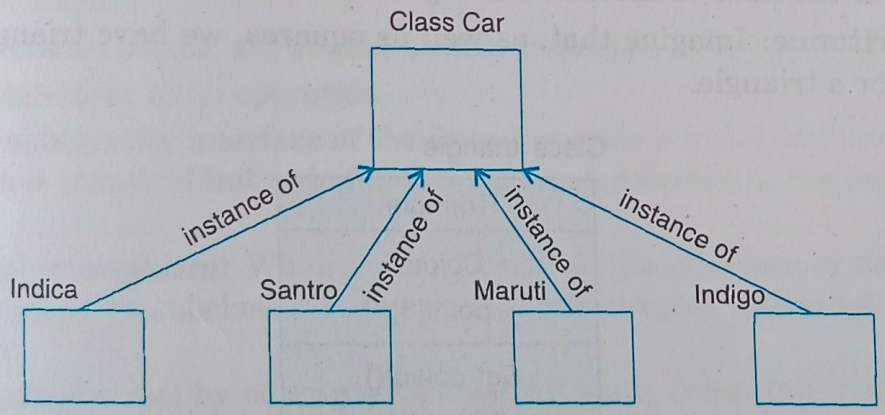


Fig. 5.20: Indica, Santro, Maruti, Indigo are all instances of the class "car"

We may have different types of classes depending upon common characteristics. The class is the static description and the object is an instance in runtime of that class.

Imagine a picture made up of squares. Each square is an object. It has a state: its colour and position, and behaviour. We can, amongst other things, change its colour and draw it. Each square is different but has much in common with other squares. So, we abstract out the commonalities: they share the same behaviour and have same sort of attributes. We have ignored same sort of attributes. We have ignored the particular values of the attributes.

Classes are useful because they act as a blue print for objects. If we want a new square we may use the square class and simply fill in the particular details (*i.e.*, colour and position). Fig. 5.21 shows how can we represent the square class.

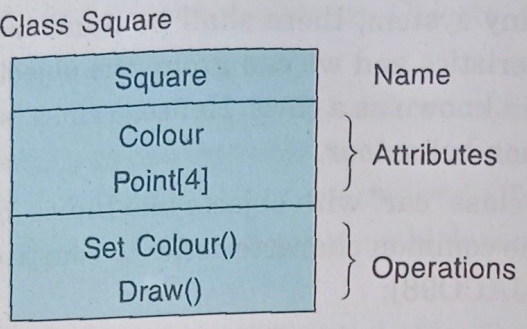


Fig. 5.21: The square class

(*v*) **Attributes:** An attribute is a data value held by the objects in a class. The square class has two attributes: a colour and array of points. Each attribute has a value for each object instance. For example, colour may be different in different objects and "array of points" size may also be different in different squares. The attributes are shown as second part of the class as shown in Fig. 5.21.

(*vi*) **Operations:** An operation is a function or transformation that may be applied to or by objects in a class. In the square class, we have two operations: set colour() and draw(). All objects in a class share the same operations. Each operation has a target object as an implicit argument. The behaviour of the operation depends on the class of its target. An object "knows" its class, and hence the right implementation of the operation [JOSHO3]. Operations are shown in the third part of the class as indicated in Fig. 5.21.

(*vii*) **Inheritance:** Imagine that, as well as squares, we have triangle class. Fig. 5.22 shows the class for a triangle.

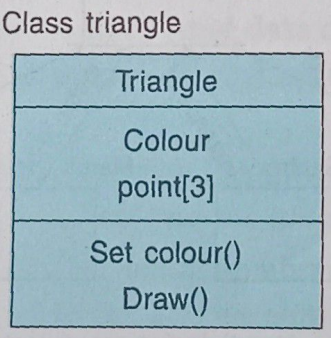


Fig. 5.22: The triangle class

Now, comparing Fig. 5.21 and Fig. 5.22, we can see that there is some difference between triangle and squares classes. Triangles have three vertices ; squares have four. Also, the way that these shapes are drawn is different. However, there are some similarities. For example, we can set the colour of both and both can be drawn (even if the way they are drawn is different). It would be nice if we could abstract; eliminate the details of each shape and amplify the fact that both can have their colours set and can be drawn. For example, at a high level of abstraction, we might want to think of a picture as made up of shapes and to draw the picture, we draw each shape in turn. We want to eliminate the irrelevant details: we do not care that one shape is a square and the other is a triangle as long as both can draw themselves.

To do this, we consider the important parts out of these classes in to a new class called Shape. Fig. 5.23 shows the results.

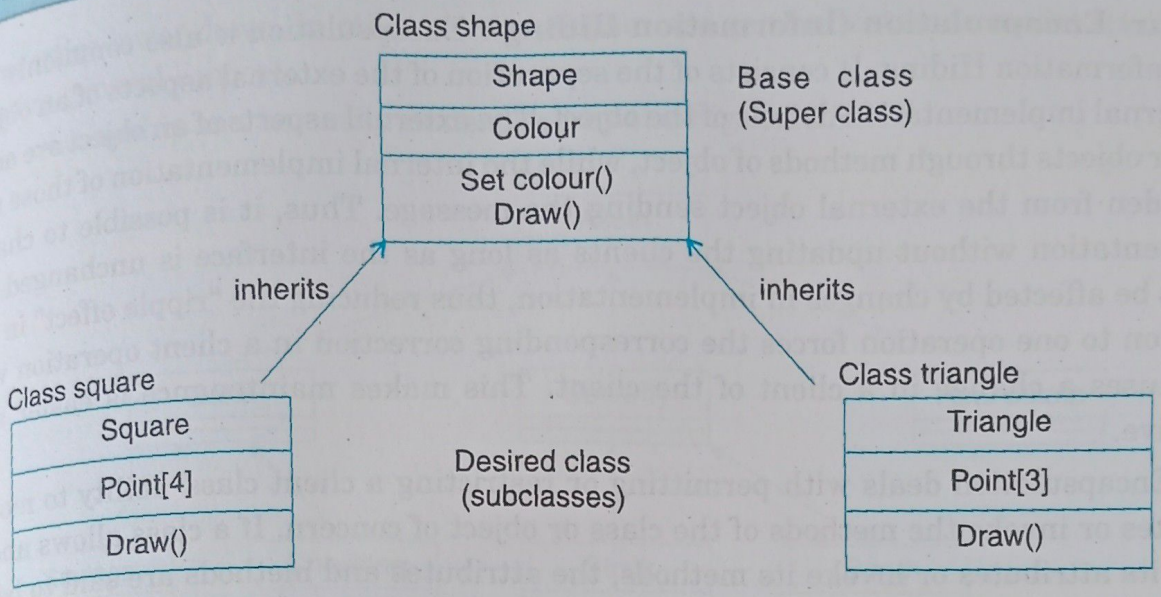


Fig. 5.23: Abstracting common features in a new class

This sort of abstraction is called inheritance. The low level classes (known as subclasses or derived classes) inherit state and behaviour from this high level class (known as a super class or base class).

Hence, a triangle object will have a colour, a setcolour() operation, three point and a draw() operation. We can inherit three sorts of things:

(*i*) State: colour

(*ii*) Operations: setcolour() should be able to set the colour for any shape.

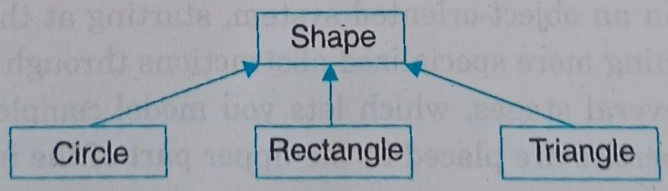
(*iii*) The interface of an operation.

Shape contains the interface of the draw() operation because draw() interface for triangle and square is identical but code for the operation remains in the subclasses because it is different.

(*viii*) **Polymorphism:** When we abstract just the interface of an operation and leave the implementation to subclasses it is called a polymorphic operation and process is called polymorphism.

So, we can abstract by pulling out important state, behaviour and interface into an new class. We can also abstract by combining object's inside a new object. In the car example, a car combines a battery, engine and other objects into a new object and provides a simple interface for driving that hides these details. There are different sorts of abstraction and finding the best ways to apply abstraction to a problem is what design is all about.

Another example may be for a base class *shape,* polymorphism enables the programmer to define different *area* methods for any number of derived classes, such as circles, rectangles and triangles. No matter what shape an object is, applying the *area* method to it will return the correct results. Plolymorphism is considered to be a requirement of any true object-oriented programming language (OOPL).



(*ix*) **Encapsulation (Information Hiding):** Encapsulation is also commonly referred to as "Information Hiding. It consists of the separation of the external aspects of an object from the internal implementation details of the object. The external aspects of an object are accessible by other objects through methods of object, while the internal implementation of those methods are hidden from the external object sending the message. Thus, it is possible to change the implementation without updating the clients as long as the interface is unchanged. Clients will not be affected by changes in implementation, thus reducing the "ripple effect" in which a correction to one operation forces the corresponding correction in a client operation which in turn causes a change in a client of the client. This makes maintenance is easier and less expensive.

Encapsulation deals with permitting or restricting a client class' ability to modify the attributes or invoke the methods of the class or object of concern. If a class allows another to modify its attributes or invoke its methods, the attributes and methods are said to be part of the class' public interface. If a class doesn't allow another to modify its attributes or invoke its methods, those are part of the class' private implementation. Thus, Encapsulation protects (a) an object's internal state from being corrupted by its clients and (b) Client code from changes in the object's implementation.

A "Queue" provides a good example of this characteristic. A queue is an abstract concept that represents an ordered list of things. The implementation of a queue may by an array or it may be by a linked-list. If the implementation were known, a developer writing a client of the queue class may use this knowledge and directly access the internal storage mechanism. If the implementation changed, the client would then have to be modified also. This type of tight coupling between classes would cause a very brittle system and would increase maintenance costs as parts of the system were modified. Therefore, the levels of encapsulation that a language supports and how those mechanisms are used directly impacts the level of coupling between classes and it can significantly affect the cost of maintenance in an application.

(*x*) **Hierarchy:** Hierarchy involves organizing something according to some particular order or rank (*e.g.*, complexity, responsibility, etc.). It is another mechanism for reducing the complexity of software by being able to treat and express sub-types in a generic way. This hierarchy is implemented in software via a mechanism called "Inheritance". Just as a child inherits genes from its parent, a class can inherit attributes and behaviours from its parent. The parent class is commonly referred to as the supper-class and the child class as the sub-class. *Classes at the same level of the hierarchy should be at the same level of abstraction.*

Using a hierarchy to describe differences or variations of a particular concept provides for more descriptive and cohesive abstractions, as well as a better allocation of responsibility. In any one system, there may be multiple abstraction hierarchies (*e.g.*, in a financial application, you may have different types of customers and accounts). Generalization can be used to realize a hierarchy within an object-oriented system, starting at the most general of the abstractions, and then defining more specialized abstractions through sub-classing. Generalization can take place in several stages, which lets you model complex, multilevel inheritance hierarchies. General properties are placed in the upper part of the inheritance hierarchy, and

special properties lower down. In other words, you can use generalization to model specialization of a more general concept.

This relationship can be easily understood by the Fig. 5.24 below.

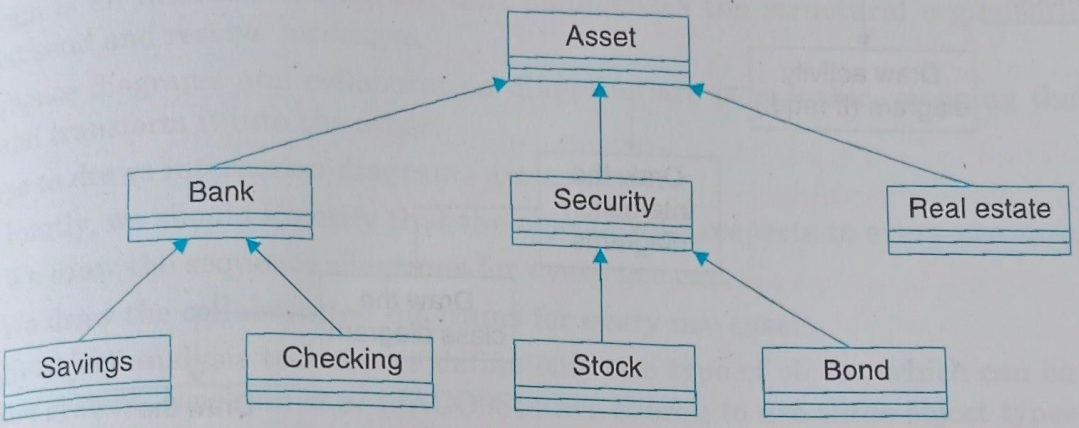


Fig. 5.24: Hierarcy

In summary, with object-oriented design, we use abstraction to break a problem into manageable chunks. We can comprehend the problem as a whole or study parts of the problem at lower level of abstraction.

If we work at an appropriate level of abstraction then any problems we find can be solved relatively easily. As an example of not working at the right level of abstraction, imagine that we tried to build a house by going out with some bricks and just building. The chances are we would finish, try to put the bath in and discover that the bathroom is too small. So, we have to knock down and rebuild a wall. Meaning, thereby, lot of work, assuming it could be done. If we had designed the house including, where the fittings were going. We would have noticed the problem and fixing it would have taken about 30 seconds with our eraser and pencil.

5.6.2 Steps to Analyze and Design Object Oriented System

There are various steps in the analysis and design of an object oriented system and are given in Fig. 5.25.

(*i*) **Create use case model:** First step is to identify the actors interacting with the system. We should then write the use case and draw the use case diagram.

(*ii*) **Draw activity diagram (If required):** Activity diagram illustrate the dynamic nature of a system by modeling the flow of control from activity to activity. An activity represents an operation on some class in the system that results in a change in the state of the system. It is essentially like a flow chart. Fig. 5.26 shows the activity diagram processing an order to deliver some goods.

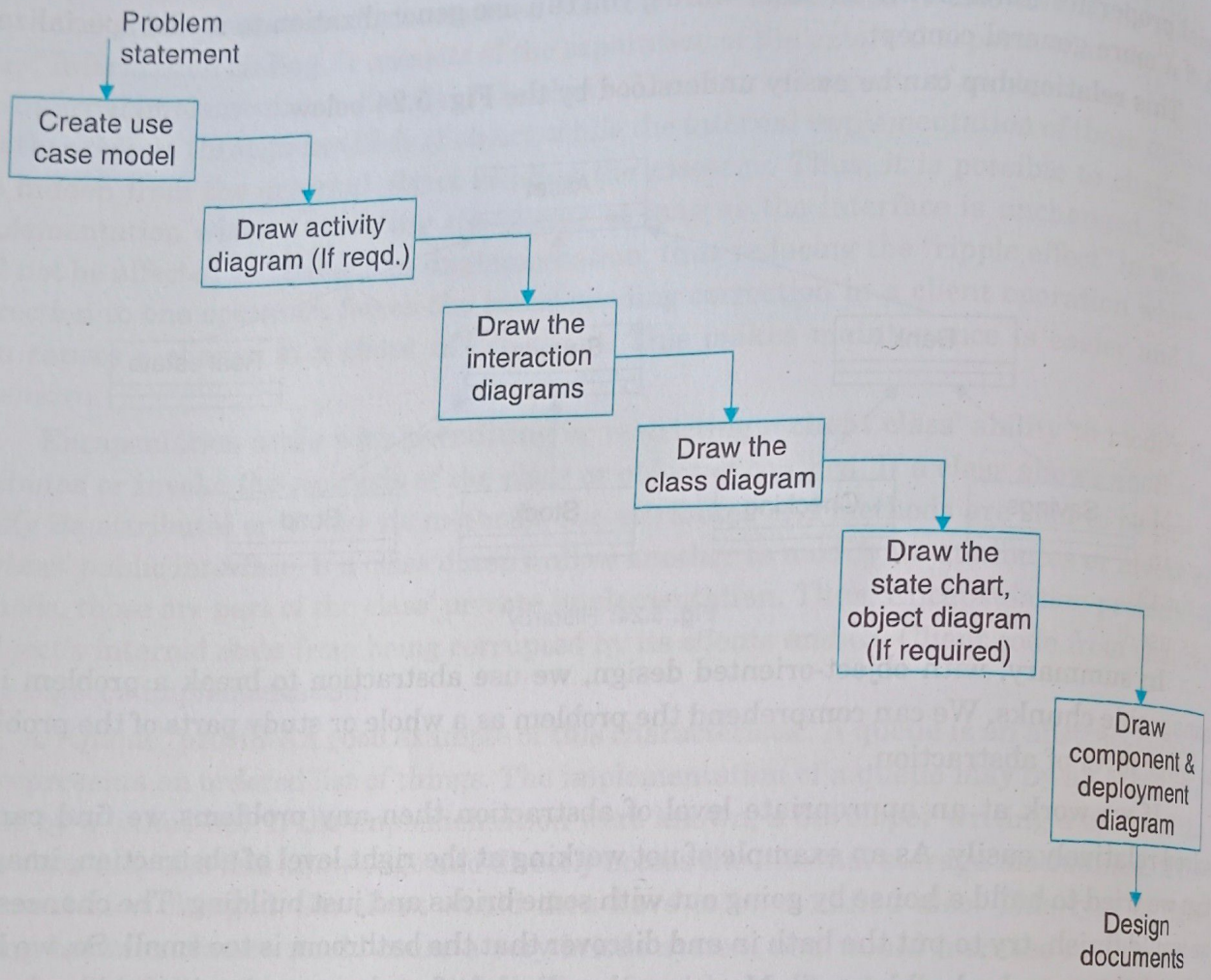


Fig. 5.25: Steps for analysis & design of object oriented system

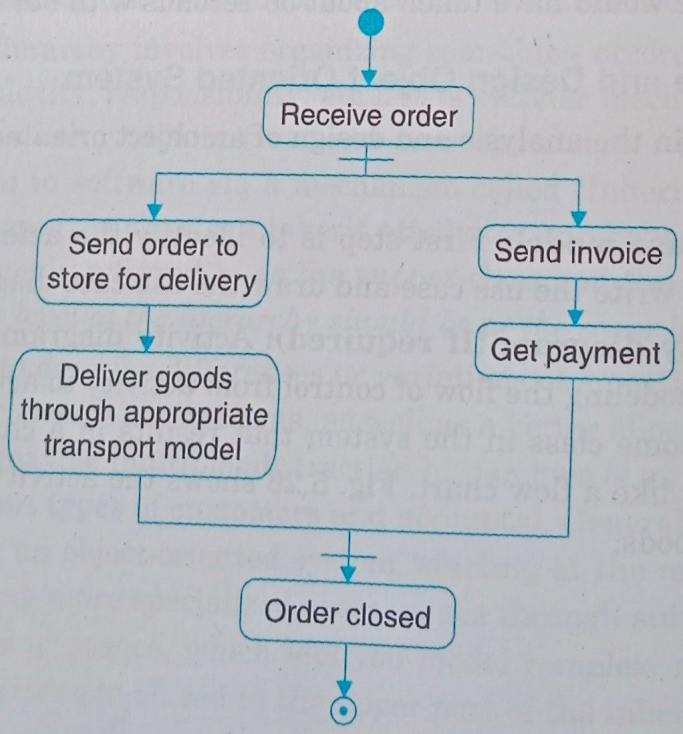


Fig. 5.26: Activity diagram

(iii) **Draw the interaction diagram:** An interaction diagram shows an interaction, consisting of a set of objects and their relationships, including the messages that may be dispatched among them. Interaction diagrams address the dynamic view of a system. A sequence diagram is an interaction diagram that emphasizes the time ordering of messages; a collaboration diagram is an interaction diagram that emphasizes the structural organisation of the objects that send and receive messages.

Sequence diagrams and collaboration diagrams are isomorphic, meaning that we can take one and transform it into the other.

Steps to draws interaction diagrams are as under :

(a) Firstly, we should identify that the objects with respects to every use case.

(b) We draw the sequence diagrams for every use case.

(c) We draw the collaboration diagrams for every use case.

Many object analysis techniques define only one type of object, which can be used any where in the system. Jacobson *et.al* [JACO98] have chosen to use three object types. The reason for this is to have a structure that is more flexible and adaptable to changes. The object types used is this analysis model are entity objects, interface objects and control objects as given in Fig. 5.27.

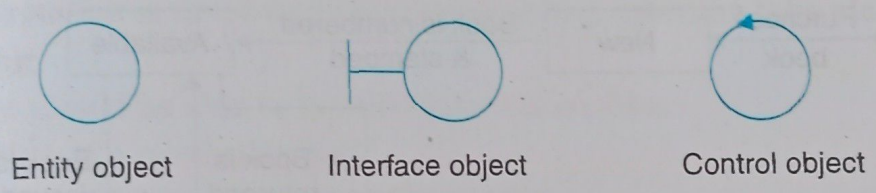


Fig. 5.27: Object types

The entity object models information in the system that should be held for a longer time, and should typically survive a use case. All behaviour naturally coupled in this information should be placed in the entity object. For a login use case, login-detail may be the entity object. The interface object models behaviour and information that is dependent on the interface to the system. Thus everything concerning any interface to the system is placed in an interface object. For example login screen of the login use case.

(iv) **Draw the class diagram:** The class diagram shows the relationship amongst classes. There are four types of relationships in class diagrams.

(a) ***Association*** are semantic connection between classes. When an association connects two classes, each class can send messages to the other in a sequence or a collaboration diagram. Associations can be bi-directional or unidirectional.

(b) ***Dependencies*** connect two classes. Dependencies are always unidirectional and show that one class, depends on the definitions in another class.

- - - - ->

(c) ***Aggregations*** are stronger form of association. An aggregation is a relationship between a whole and its parts.

(*d*) ***Generalizations*** are used to show an inheritance relationship between two cla

We may design the class diagram of a complete system. Obviously, this may bec very huge, for complex systems. In such a case, one class diagram is made for each use This restricts the size of the class diagram and is useful when delegating/assigning the tas a software developer ; since he/she would only need to see the related class diagram of assigned use case; rather than looking at the complete class diagram and then filtering out concepts related to his/her use-case. We may also draw object diagram, which shows a se objects and their relationships. They represent static snapshots of instances of the things fo in class diagrams, from the perspective of real cases.

(*v*) ***Design of state chart diagrams:*** A state chart diagram is used to show the st space of a given class, the event that cause a transition from one state to another, and actions that result from a state change. It shows the changing behaviour of an object in sponse to different events. State transition diagrams are made only for the objects that either very complicated or have a very dynamic behaviour with respect to various even Normally, we would not need to make state diagrams. A state transition diagram for a "boo in the library system is given in Fig. 5.28.

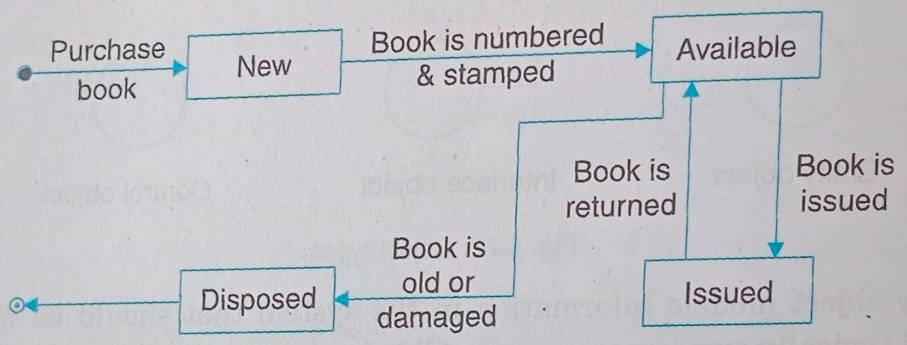


Fig. 5.28: Transition chart for 'book' in a library system

(*vi*) ***Draw component and development diagram:*** Component diagrams address the static implementation view of a system they are related to class diagrams in that a component typically maps to one or more classes, interfaces or collaboration. Deployment diagram captures relationship between physical components and the hardware.

After the completion of above mentioned steps, we will have many documents and complete understanding of flow of data, control and relationships of classes. These things are the foundations for implementing an object oriented system.

5.6.3 Case Study of Library Management System

The problem statement for library management system is given below:

Problem statement

A software has to be developed for automating the manual library of a University. The system should be stand alone in nature. It should be designed to provide functionality's as explained below:

Issue of books

- A student of any course should be able to get books issued.
- Books from general section are issued to all but book bank books are issued only for their respective courses.
- A limitation is imposed on the number of books a student can issue.
- A maximum of 4 books from book bank and 3 books from general section is issued for 15 days only.
- The software takes the current system date as the date of issue and calculates date of return.
- A bar code detector is used to save the student as well as book information.
- The due date for return of the book is stamped on the book.

Return of books

- Any person can return the issued books.
- The student information is displayed using the bar code detector.
- The system displays the student details on whose name the books were issued as well as the date of issue and return of the book.
- The system operator verifies the duration for the issue.
- The information is saved and the corresponding updating take place in the database.

Query processing

The system should be able to provide information like:

- Availability of a particular book.
- Availability of book of any particular author.
- Number of copies available of the desired book.

The system should also be able to generate reports regarding the details of the books available in the library at any given time. The corresponding printouts for each entry (issue, return) made in the system should be generated. Security provisions like the 'login authenticity should be provided. Each user should have a user id and a password. Record of the users of the system should be kept in the log file. Provision should be made for full backup of the system.

Use cases

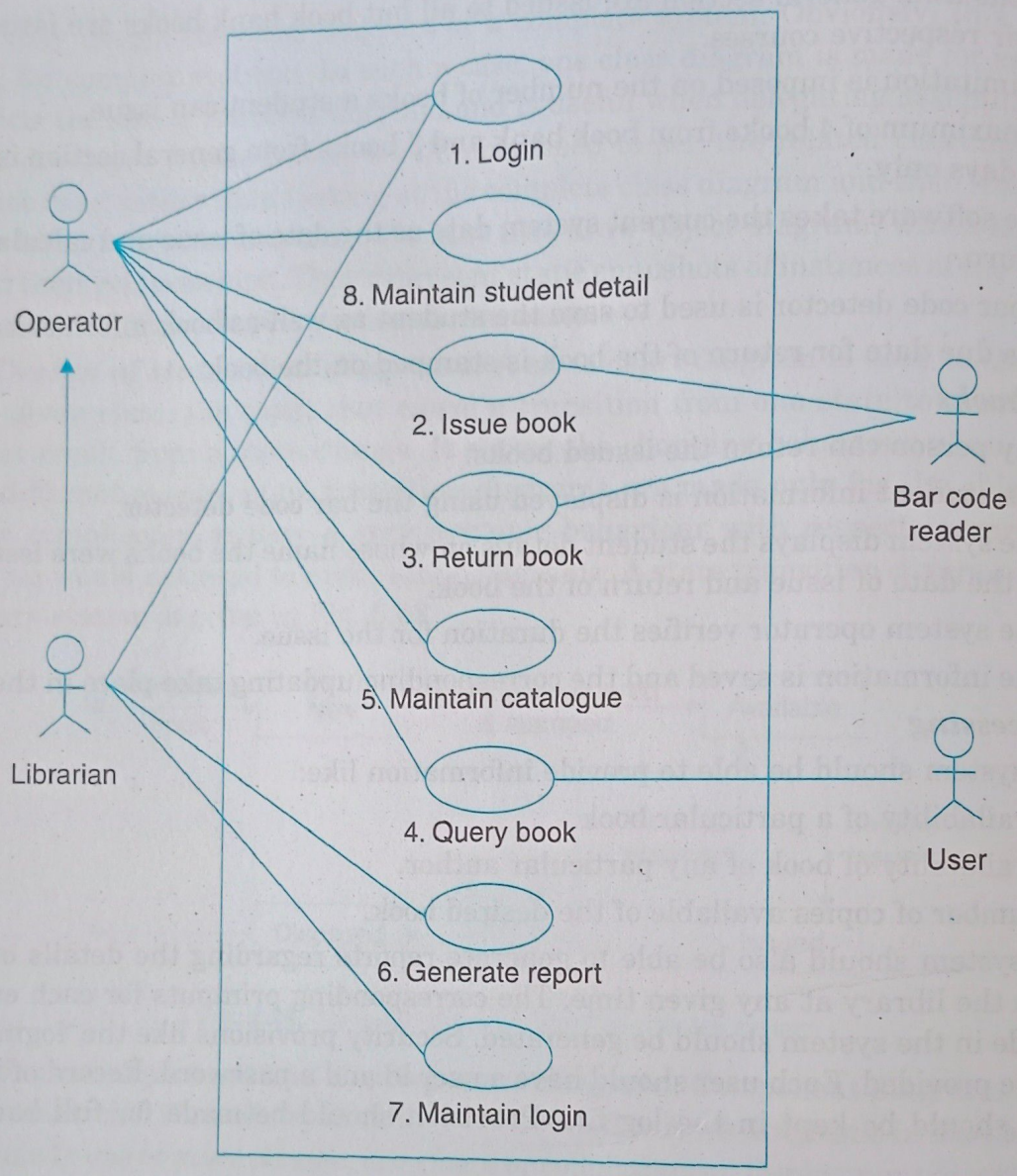
From the problem description, we can see that the system has four actors.

1. Librarian
2. Operator
3. Bar Code Reader
4. User

To define system's functionalities, we can view the system as a collection of following use cases:

1. Login
2. Issue Book
3. Return Book
4. Query Book

5. Maintain Catalog
6. Generate Report
7. Maintain Login
8. Maintain student Details



Use case diagram for library management system

USE CASE DESCRIPTION

1. LOGIN

1.1. Introduction

This use case documents the procedure for logging into the Library Management System based on user privileges.

- Operator (Issue Book, Return Book, Query a Book, Maintain Catalogue, Maintain Student Detail)
- Librarian (Generate Reports, Maintain Login)

1.2 Actors

Operator, Librarian

1.3 Pre-Condition

None

1.4 Post-Condition

If use case is successful, the user is logged into the system, otherwise the system state is unchanged.

1.5 Flow of Events

1.5.1 Basic Flow

This use case starts when actor wishes to log in to the Library Management System.

- The system requests that the actor enters his/her user_id and password.
- The actor enters user_id and password.
- The system validates the user_id and password and checks for his/her privileges.
- If the user is "operator", he/she will be logged into the system and presented with operator's menu.
- Otherwise, if the user is "librarian", he/she will be logged into the system and presented with librarian's menu.
- The use case ends.

1.5.2 Alternate Flow

1.5.2.1 Invalid Name/Password

If the system receives an invalid user_id or password, and error message is displayed and the use case ends.

1.6 Special Requirements

None

1.7 Related Use Cases

None

2. ISSUE BOOK

2.1 Introduction

This use case documents the procedure of issuing a book for following accounts:

- General (for 15 days)
- Book Bank (for the semester)

2.2 Actors

Operator, Barcode reader

2.3 Pre-Condition

Operator must be logged in to the system

2.4 Post-Condition

If use case is successful, the book is issued to the student in his/her general or book bank account, otherwise the system state is unchanged.

2.5 Flow of Events

2.5.1 Basic Flow

The use case starts when a student wants to get a book issued.

- The system reads and validates the student's information using the Bar Code Reader.
- The system reads the book's information using the Bar Code Reader.
- The return date of the book is calculated as per the account in which the student wishes to get the book issued – 15 days for General account and whole semester for Book Bank account.
- The book and student's information is saved into the database.
- The issue details are sent to the printer to generate the receipt.
- The use case ends.

2.5.2 *Alternate Flow*

2.5.2.1 Unauthorized Student

If the system doesn't validates the student, then an error message is flagged and the use case ends.

2.5.2.2 General Account is Full

If the student has requested a book in General Account and the later is full, *i.e.* he has already 3 books issued on his name, then the request for issue is denied and the use case ends.

2.5.2.3 Book Bank Account is Full

If the student has requested a book in Book Bank Account and the later is full, *i.e.* he has already 4 books issued on his name, then and error message is shown and the use case ends.

2.5.2.4 Course Mismatch between Student and the Book Bank Book.

If the student of a particular course has requested a book from some other course on Book Bank Account, the an option for getting the book issued in general account is given and the use case ends.

2.6 Special Requirements

None

2.7 Related Use Cases

Generate Barcode

3. RETURN BOOK

3.1 Introduction

This use case documents the procedure of returning a book and calculating the fine amount if the students has returned the book after the specified return date.

3.2 Actors

Operator, Barcode Reader

3.3 Pre-Condition

Operator must be logged in to the system.

3.4 Post-Condition

If use case is successful, the book is returned back to the library and if needed, the fine is calculated, otherwise the system state is unchanged.

3.5 Flow of Events

3.5.1 Basic Flow

This use case starts when a student wants to return a book.

- The system reads the book's information using the Bar Code Reader.
- The book is returned to the library.
- The database entries corresponding both to the student account and the book are updated.
- The return details are sent to the printer to generate the receipt.
- The use case ends.

3.5.2 Alternate Flow

3.5.2.1 Late return of book

If the book is returned after the due date, fine is calculated and database is updated accordingly. The use case end here.

3.6 Special Requirements

None

3.7 Related Use Cases

Generate Barcode

4. QUERY A BOOK

4.1 Introduction

This use case documents the procedure for searching a book based on the specified criteria, which are:

- Search by Author Name
- Search by Title Name

4.2 Actors

Operator, user

4.3 Pre-Condition

Operator user must be logged in to the system

4.4 Post-Condition

If use case is successful, the book details are displayed.

4.5 Flow of Events

4.5.1 Basic Flow

This use case starts when a student wants to search for a particular book.

- The system displays the various search criteria to the user.
- The user selects the search criteria.
- The result is displayed to the user.
- The use case ends.

4.6 Special Requirements

None

4.7 Related Use Cases

None

5. MAINTAIN CATALOG

5.1 Introduction

This use case documents the procedure for updating the catalog of the library.

5.2 Actors

Operator

5.3 Pre-Condition

Operator must be logged into the system.

5.4 Post-Condition

If use case is successful, the book should be updated, otherwise the system state in unchanged.

5.5 Flow of Events

5.5.1 Basic Flow

This use case starts when the operator wishes to add, delete or modify some details in the library.

- The corresponding changes are saved in the database.
- The use case ends.

5.5.2 Alternate Flow

None

5.6 Special Requirements

None

5.7 Related Use Cases

None

6. GENERAL REPORTS

6.1 Introduction

This use case documents the procedure for generating the reports as desired by the Librarian.

6.2 Actors

Librarian

6.3 Pre-Condition

Librarian must be logged into the system.

6.4 Post-Condition

If use case is successful, the various reports, regarding the details of the books available in the library at any given time, are generated.

6.5 Flow of Events

6.5.1 Basic Flow

This use case starts when a librarian wants to generate reports of the books available in the library.

- The system displays the various report generating criteria to the user, which can be the books issued to the students at a particular time, books available in the library etc.

- The librarian selects the criteria and enters the various parameters based on the criteria selected.
- The system generates the report and sends that to printer.
- The use case ends.

6.5.2 *Alternate Flow*

6.5.2.1 Printer out of paper or low on ink

If the printer goes out of paper or low on ink, then the printing operation is aborted and the necessary action needs to be taken, which can be feeding paper to the printer or replacing the ink cartridge. The use case ends.

6.6 Special Requirements

None

6.7 Related Use Cases

None

MAINTAIN LOGIN

7.1 Introduction

This use case documents the procedure for maintaining Login Details.

7.2 Actors

Librarian.

7.3 Pre-Condition

Librarian must be logged into the system.

7.4 Post-Condition

If use case is successful, the Login details should be updated, otherwise the system state is unchanged.

7.5 Flow of Events

7.5.1 *Basic Flow*

This use case starts when the Librarian wishes to add, delete or modify some details of login.

- The corresponding changes will be done.
- The use case ends.

7.5.2 *Alternate Flow*

None

7.6 Special Requirements

None

7.7 Related Use Cases

None

MAINTAIN STUDENT DETAILS

8.1 Introduction

This use case documents the procedure for maintaining student details.