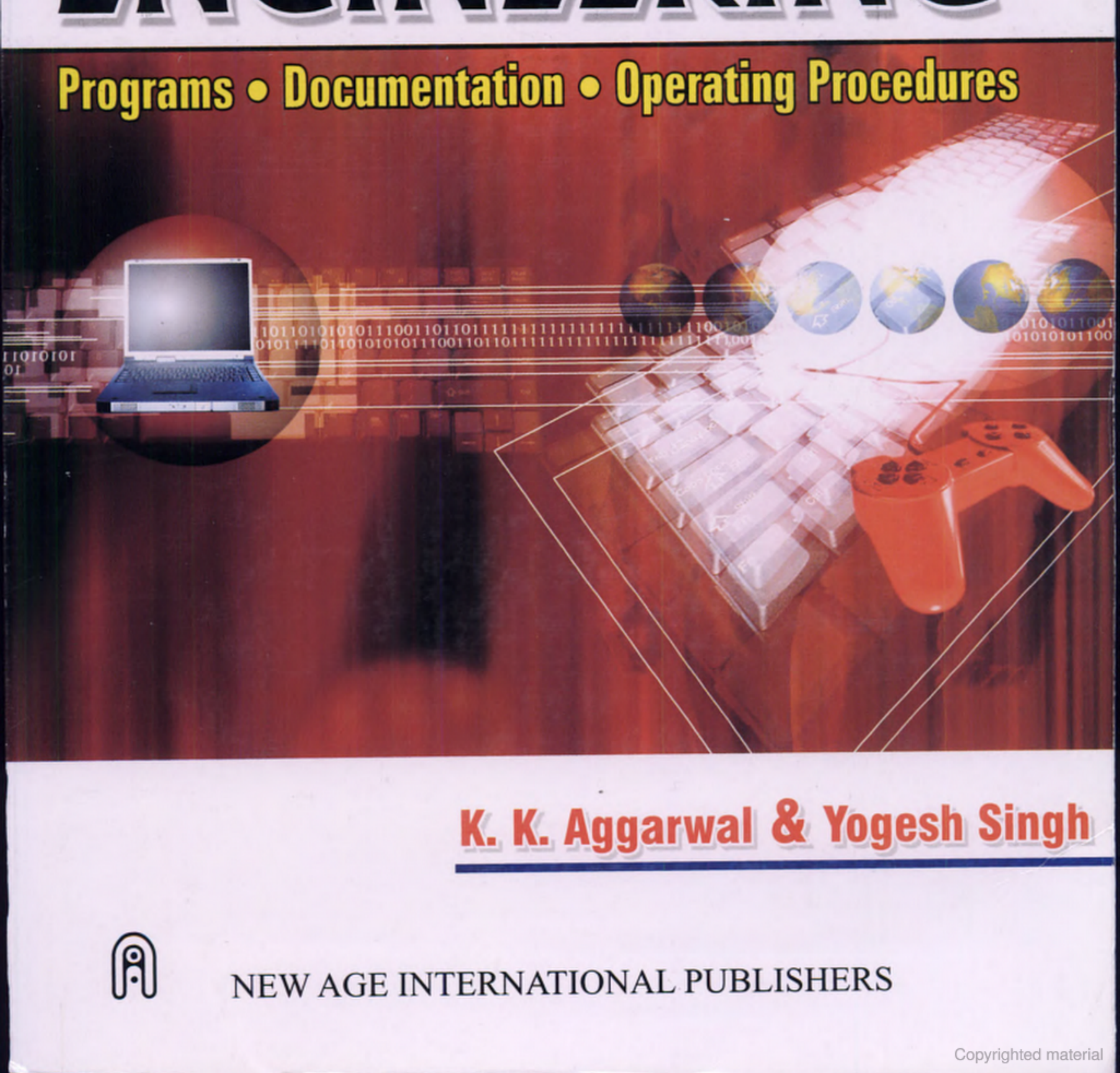
Revised Second Edition

SOFTWARE ENGINEERING

Programs • Documentation • Operating Procedures



K. K. Aggarwal & Yogesh Singh

NEW AGE INTERNATIONAL PUBLISHERS

SOFTWARE ENGINEERING

Programs • Documentation • Operating Procedures

Revised Second Edition

K. K. Aggarwal
Professor and Vice-Chancellor
Guru Gobind Singh Indraprastha University, Delhi

Yogesh Singh
Professor and Dean
University School of Information Technology
Guru Gobind Singh Indraprastha University, Delhi

NEW AGE

NEW AGE INTERNATIONAL (P) LIMITED, PUBLISHERS
New Delhi • Bangalore • Chennai • Cochin • Guwahati • Hyderabad
Jalandhar • Kolkata • Lucknow • Mumbai • Ranchi

First Edition 2001
Revised Second Edition 2005

NEW AGE INTERNATIONAL (P) LIMITED, PUBLISHERS
4835/24, Ansari Road, Daryaganj,
New Delhi - 110 002
Visit us at : www.newagepublishers.com

Offices at :

Bangalore, Chennai, Cochin, Guwahati, Hyderabad, Jalandhar,
Kolkata, Lucknow, Mumbai and Ranchi

Rs. 215.00

ISBN : 81-224-1638-1

Published by New Age International (P) Ltd.,
4835/24, Ansari Road, Daryaganj, New Delhi-110 002 and
printed in India at Pack Printers, New Delhi
Typesetter : Goswami Printers, Delhi-110 053

1 Introduction

Contents

1 Introduction

The nature and complexity of software have changed significantly in the last 30 years. In the 1970s, applications ran on a single processor, produced alphanumeric output, and received their input from a linear source. Today's applications are far more complex; typically have graphical user interface and client-server architecture. They frequently run on two or more processors, under different operating systems, and on geographically distributed machines.

Rarely, in history has a field of endeavor evolved as rapidly as software development. The struggle to stay, abreast of new technology, deal with accumulated development backlogs, and cope with people issues has become a treadmill race, as software groups work as hard as they can, just to stay in place. The initial concept of one "guru", indispensable to a project and hostage to its continued maintenance has changed. The Software Engineering Institute (SEI) and group of "gurus" advise us to improve our development process. Improvement means "ready to change". Not every member of an organization feels the need to change. It is too easy to dismiss process improvement efforts as just the latest management fad. Therein lie the seeds of conflict, as some members of a team embrace new ways of working, while others mutter "over my dead body" [WIEG94].

Therefore, there is an urgent need to adopt software engineering concepts, strategies, practices to avoid conflict, and to improve the software development process in order to deliver good quality maintainable software in time and within budget.

1.1 SOFTWARE CRISIS

The software crisis has been with us since 1970. Since then, the computer industry has progressed at a break-neck speed through the computer revolution, and recently, the network revolution triggered and/or accelerated by the explosive spread of the internet and most recently the web. Computer industry has been delivering exponential improvement in price-performance, but the problems with software have not been decreasing. Software still come late, exceed budget and are full of residual faults. As per the latest IBM report, "31% of the projects get cancelled before they are completed, 53% over-run their cost estimates by an average of 189% and for every 100 projects, there are 94 restarts" [IBMG2K]. History has seen many software failures. Some of these are :

(*i*) The Y2K problem was the most crucial problem of last century. It was simply the ignorance about the adequacy or otherwise of using only last two digits of the year. The 4-digit date format, like 1964, was shortened to 2-digit format, like 64. The developers could not

visualise the problem of year 2000. Millions of rupees have been spent to handle this practically non-existent problem.

(*ii*) The "star wars" program of USA produced "Patriot missile" and was used first time in Gulf war. Patriot missiles were used as a defence for Iraqi Scud missiles. The Patriot missiles failed several times to hit Scud missiles, including one that killed 28 U.S. soldiers in Dhahran, Saudi Arabia. A review team was constituted to find the reason and result was software bug. A small timing error in the system's clock accumulated to the point that after 14 hours, the tracking system was no longer accurate. In the Dhahran attack, the system had been operating for more than 100 hours.

(*iii*) In 1996, a US consumer group embarked on an 18-month, $1 million project to replace its customer database. The new system was delivered on time but did not work as promised, handling routine transactions smoothly but tripping over more complex ones. Within three weeks the database was shutdown, transactions were processed by hand and a new team was brought in to rebuild the system. Possible reasons for such a failure may be that the design team was over optimistic in agreeing to requirements and developers became fixated on deadlines, allowing errors to be ignored.

(*iv*) "One little bug, one big crash" of Ariane-5 space rocket, developed at a cost of $7000 M over a 10 year period. The space rocket was destroyed after 39 seconds of its launch, at an altitude of two and a half miles alongwith its payload of four expensive and uninsured scientific satellites. The reason was very simple. When the guidance system's own computer tried to convert one piece of data—the sideways velocity of the rocket—from a 64-bit format to a 16-bit format; the number was too big, and an overflow error resulted after 36.7 seconds. When the guidance system shutdown, it passed control to an identical, redundant unit, which was there to provide backup in case of just such a failure. Unfortunately, the second unit had failed in the identical manner a few milliseconds before. In this case, the developers had decided that this particular velocity figure would never be large enough to cause trouble—after all, it never had been before.

We may discuss many such failures which have played with human safety and caused the project to fail in past. Hence, in order to handle such unfortunate events, a systematic and scientific discipline is required and this emerging discipline is software engineering.

1.1.1 No Silver Bullet

As we all know, the hardware cost continues to decline drastically. However, there are desperate cries for a silver bullet-something to make software costs drop as rapidly as computer hardware costs do. But as we look to the horizon of a decade, we see no silver bullet. There is no single development, either in technology or in management technique, that by itself promises even one order-of-magnitude improvement in productivity, in reliability, in simplicity.

Inventions in electronic design through transistors and large scale integration has significantly affected the cost, performance and reliability of the computer hardware. No other technology, since civilization began, has seen six orders of magnitude in performance-price gain in 30 years. The progress in software technology is not that rosy due to certain difficulties with this technology. Some of the difficulties are complexity, changeability and invisibility.

The hard part of building software is the specification, design and testing of this conceptual construct, not the labour of representing it and testing the correctness of

representation. We still make syntax errors, to be sure, but they are trivial as compared to the conceptual errors (logic errors) in most systems. That is why, building software is always hard and there is inherently no silver bullet.

Many people (especially CASE tool vendors) believe that CASE (Computer Aided Software Engineering) tools represent the so-called silver bullet that would rescue the software industry from the software crisis. Many companies have used these tools and spent large sums of money, but results were highly unsatisfactory, we learnt the hard way that there is no such thing as a silver bullet [BROO87].

1.1.2 Software Myths

There are number of myths associated with software development community. Some of them really affect the way, in which software development should take place. In this section, we list few myths, and discuss their applicability to standard software development. [PIER99, LEVE95].

1. **Software is easy to change.** It is true that source code files are easy to edit, but that is quite different than saying that software is easy to change. This is deceptive precisely because source code is so easy to alter. But making changes without introducing errors is extremely difficult, particularly in organizations with poor process maturity. Every change requires that the complete system be re-verified. If we do not take proper care, this will be an extremely tedious and expensive process.

2. **Computers provide greater reliability than the devices they replace.** It is true that software does not fail in the traditional sense. There are no limits to how many times a given piece of code can be executed before it "wears out". In any event, the simple expression of this myth is that our general ledgers are still not perfectly accurate, even though they have been computerized. Back in the days of manual accounting systems, human error was a fact of life. Now, we have software error as well.

3. **Testing software or "proving" software correct can remove all the errors.** Testing can only show the presence of errors. It cannot show the absence of errors. Our aim is to design effective test cases in order to find maximum possible errors. The more we test, the more we are confident about our design.

4. **Reusing software increases safety.** This myth is particularly troubling because of the false sense of security that code re-use can create. Code re-use is a very powerful tool that can yield dramatic improvement in development efficiency, but it still requires analysis to determine its suitability and testing to determine if it works.

5. **Software can work right the first time.** If we go to an aeronautical engineer, and ask him to build a jet fighter craft, he will quote us a price. If we demand that it is to be put in production without building a prototype, he will laugh and may refuse the job. Yet, software engineers are often asked to do precisely this sort of work, and they often accept the job.

6. **Software can be designed thoroughly enough to avoid most integration problems.** There is an old saying among software designers: "Too bad, there is no complier for specifications": This points out the fundamental difficulty with detailed specifications. They always have inconsistencies, and there is no computer tool to perform consistency checks on these. Therefore, special care is required to understand the specifications, and if there is an ambiguity, that should be resolved before proceeding for design.

7. **Software with more features is better software.** This is, of course, almost the opposite of the truth. The best, most enduring programs are those which do one thing well.

8. **Addition of more software engineers will make up the delay.** This is not true in most of the cases. By the process of adding more software engineers during the project, we may further delay the project. This does not serve any purpose here, although this may be true for any civil engineering work.

9. **Aim is to develop working programs.** The aim has been shifted from developing working programs to good quality, maintainable programs. Maintaining software has become a very critical and crucial area for software engineering community.

This list is endless. These myths, poor quality of software, increasing cost and delay in the delivery of the software have been the driving forces behind the emergence of software engineering as a discipline. In addition, following are the contributing factors:

- Change in ratio of hardware to software costs
- Increasing importance of maintenance
- Advances in software techniques
- Increased demand for software
- Demand for larger and more complex software systems.

1.2 WHAT IS SOFTWARE ENGINEERING?

Software has become critical to advancement in almost all areas of human endeavour. The art of programming only is no longer sufficient to construct large programs. There are serious problems in the cost, timeliness, maintenance and quality of many software products.

Software Engineering has the objective of solving these problems by producing good quality, maintainable software, on time, within budget. To achieve this objective, we have to focus in a disciplined manner on both the quality of the product and on the process used to develop the product.

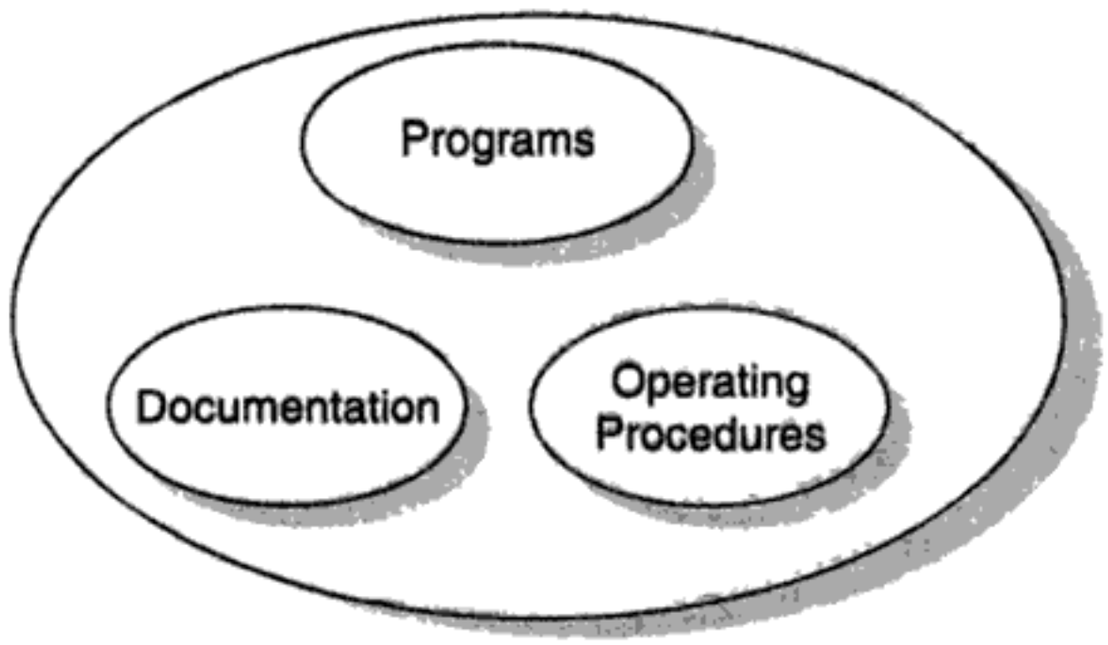
1.2.1 Definition

At the first conference on software engineering in 1968, Fritz Bauer [FRIT68] defined software engineering as *"The establishment and use of sound engineering principles in order to obtain economically developed software that is reliable and works efficiently on real machines"*. Stephen Schach [SCHA90] defined the same as *"A discipline whose aim is the production of quality software, software that is delivered on time, within budget, and that satisfies its requirements"*.

Both the definitions are popular and acceptable to majority. However, due to increase in cost of maintaining software, objective is now shifting to produce quality software that is maintainable, delivered on time, within budget, and also satisfies its requirements.

1.2.2 Program Versus Software

Software is more than programs. It consists of programs, documentation of any facet of the program and the procedures used to setup and operate the software system. The components of the software systems are shown in Fig. 1.1.



Software = Program + Documentation + Operating Procedures

Fig. 1.1: Components of software

Any program is a subset of software and it becomes software only if documentation and operating procedure manuals are prepared. Program is a combination of source code and object code. Documentation consists of different types of manuals as shown in Fig. 1.2.

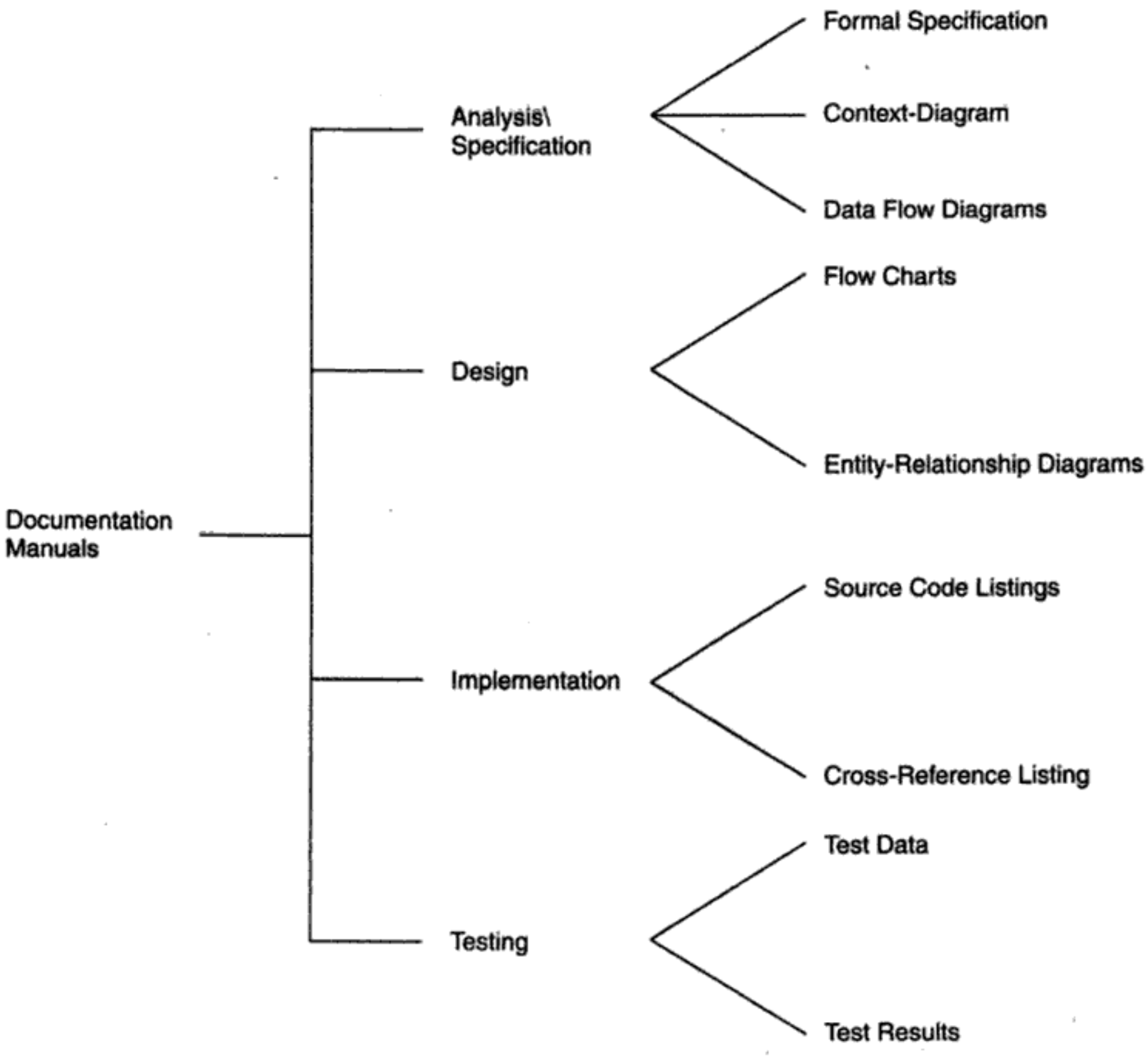


Fig. 1.2: List of documentation manuals.

Operating procedures consist of instructions to setup and use the software system and instructions on how to react to system failure. List of operating procedure manuals/documents is given in Fig. 1.3.

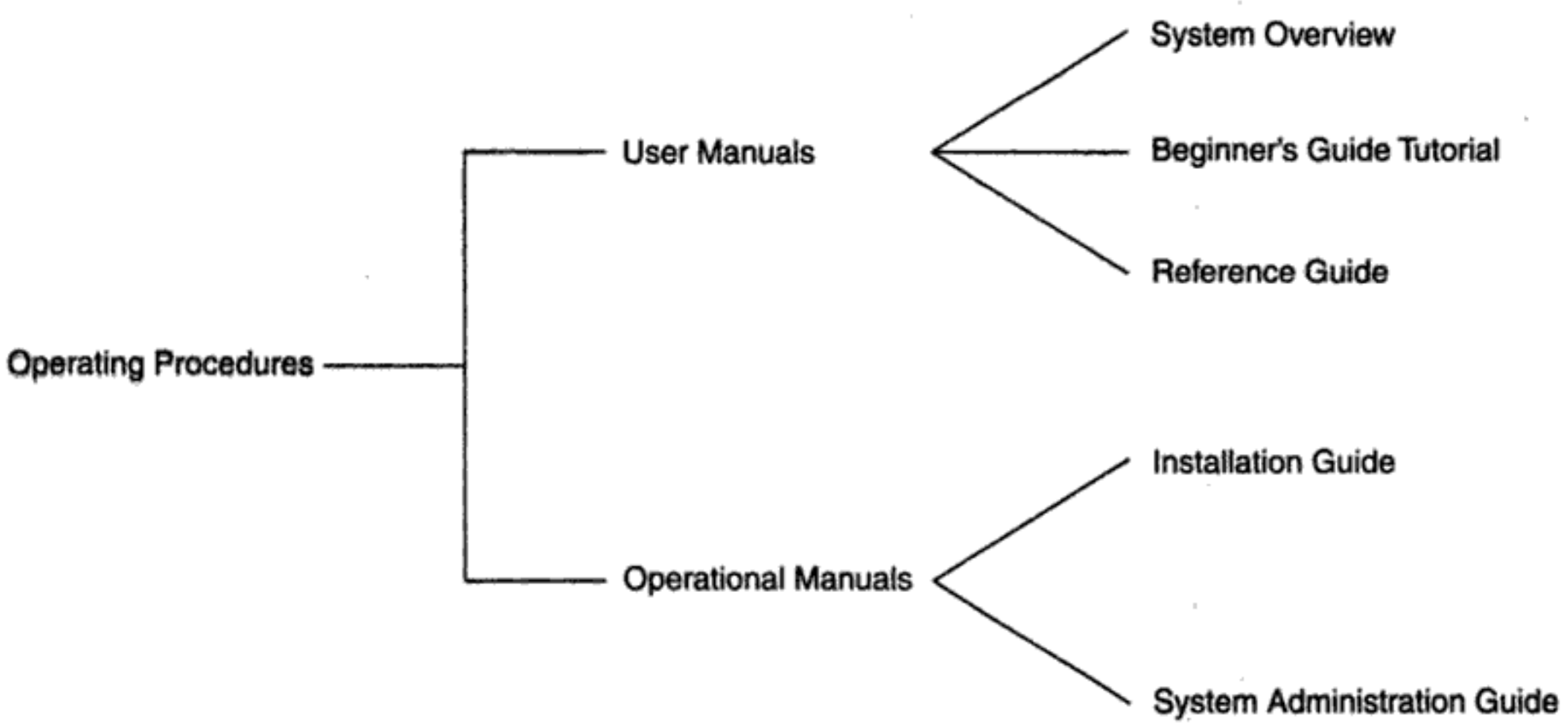


Fig. 1.3: List of operating procedure manuals.

1.2.3 Software Process

The software process is the way in which we produce software. This differs from organization to organization. Surviving in the increasingly competitive software business requires more than hiring smart, knowledgeable developers and buying the latest development tools. We also need to use effective software development processes, so that developers can systematically use the best technical and managerial practices to successfully complete their projects. Many software organizations are looking at software process improvement as a way to improve the quality, productivity, predictability of their software development, and maintenance efforts [WIEG96].

It seems straight forward, and the literature has a number of success stories of companies that substantially improved their software development and project management capabilities. However, many other organizations do not manage to achieve significant and lasting improvements in the way they conduct their projects. Here we discuss few reasons why is it difficult to improve software process [HUMP89, WIEG99] ?

1. Not enough time. Unrealistic schedules leave insufficient time to do the essential project work. No software groups are sitting around with plenty of spare time to devote to exploring what is wrong with their current development processes and what they should be doing differently. Customers and senior managers are demanding more software, of higher quality in minimum possible time. Therefore, there is always a shortage of time. One consequence is that software organizations may deliver release 1.0 on time, but then they have to ship release 1.01 almost immediately thereafter to fix the recently discovered bugs.

2. **Lack of knowledge.** A second obstacle to widespread process improvement is that many software developers do not seem to be familiar with industry best practices. Normally,

software developers do not spend much time reading the literature to find out about the best-known ways of software development. Developers may buy books on Java, Visual Basic or ORACLE, but do not look for anything about process, testing or quality on their bookshelves.

The industry awareness of process improvement frameworks such as the capability maturity model and ISO 9001 for software (discussed in Chapter 7) have grown in recent years, but effective and sensible application still is not that common. Many recognized best practices available in literature simply are not in widespread use in the software development world.

3. **Wrong motivations.** Some organizations launch process improvement initiatives for the wrong reasons. May be an external entity, such as a contractor, demanded that the development organization should achieve CMM level X by date Y. Or perhaps a senior manager learned just enough about the CMM and directed his organization to climb on the CMM bandwagon.

The basic motivation for software process improvement should be to make some of the current difficulties we experience on our projects to go away. Developers are rarely motivated by seemingly arbitrary goals of achieving a higher maturity level or an external certification (ISO 9000) just because someone has decreed it. However, most people should be motivated by the prospect of meeting their commitments, improving customer satisfaction, and delivering excellent products that meet customer expectations. The developers have resisted many process improvement initiatives when they were directed to do "the CMM thing", without a clear explanation of the reasons why improvement was needed and the benefits the team expected to achieve.

4. **Insufficient commitment.** Many times, the software process improvement fails, despite best of intentions, due to lack of true commitment. It starts with a process assessment but fails to follow through with actual changes. Management sets no expectations from the development community around process improvement; they devote insufficient resources, write no improvement plan, develop no roadmap, and pilot no new processes.

The investment we make in process improvement will not have an impact on current productivity; because the time we spend developing better ways to work tomorrow is not available for today's assignment. It can be tempting to abandon the effort when skeptics see the energy they want to be devoted to immediate demands being siphoned off in the hope of a better future (refer Fig. 1.4). Software organizations should not give up, but should take

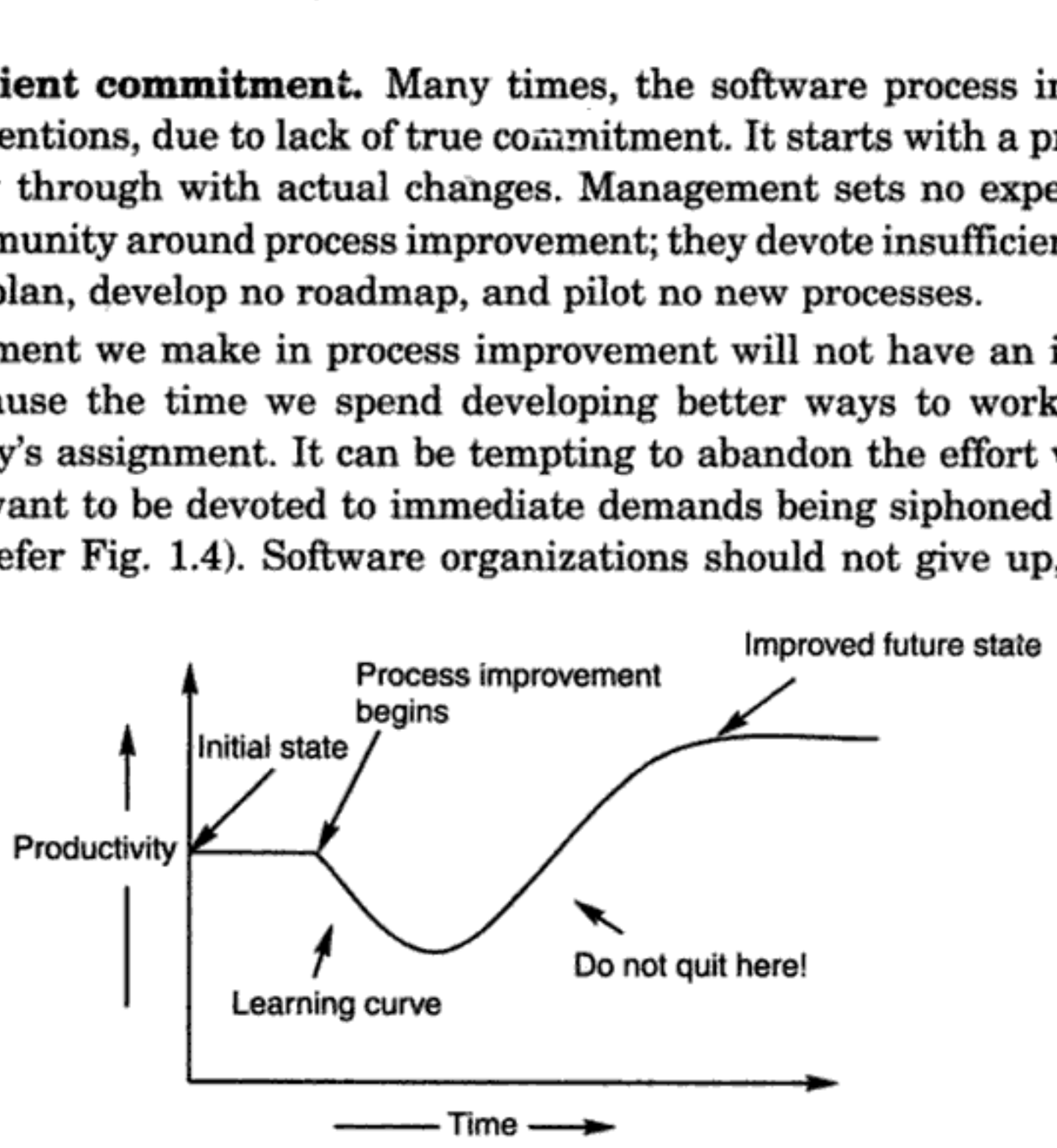


Fig. 1.4: The process improvement learning curve.

motivation from the very real, long-term benefits that many companies (including Motorola, Hewlett-Packard, Boeing, Microsoft etc.) have enjoyed from sustained software process improvement initiatives. Improvements will take place over time and organizations should not expect and promise miracles [WIEG2K] and should always remember the learning curve.

1.2.4 Software Characteristics

The software has a very special characteristic *e.g.*, "it does not wear out". Its behaviour and nature is quite different than other products of human life. A comparison with one such case, *i.e.*, constructing a bridge vis-a-vis writing a program is given in Table 1.1. Both activities require different processes and have different characteristics.

Table 1.1: A comparison of constructing a bridge and writing a program

Sr. No.	*Constructing a bridge*	*Writing a program*
1.	The problem is well understood.	Only some parts of the problem are understood, others are not.
2.	There are many existing bridges.	Every program is different and designed for special applications.
3.	The requirements for a bridge typically do not change much during construction.	Requirements typically change during all phases of development.
4.	The strength and stability of a bridge can be calculated with reasonable precision.	Not possible to calculate correctness of a program with existing methods.
5.	When a bridge collapses, there is a detailed investigation and report.	When a program fails, the reasons are often unavailable or even deliberately concealed.
6.	Engineers have been constructing bridges for thousands of years.	Developers have been writing programs for 50 years or so.
7.	Materials (wood, stone, iron, steel) and techniques (making joints in wood, carving stone, casting iron) change slowly.	Hardware and software changes rapidly.

Some of the important characteristics are discussed below:

(*i*) **Software does not wear out.** There is a well-known "bath tub curve" in reliability studies for hardware products. The curve is given in Fig. 1.5. The shape of the curve is like "bath tub"; and is known as bath tub curve.

There are three phases for the life of a hardware product. Initial phase is burn-in phase, where failure intensity is high. It is expected to test the product in the industry before delivery. Due to testing and fixing

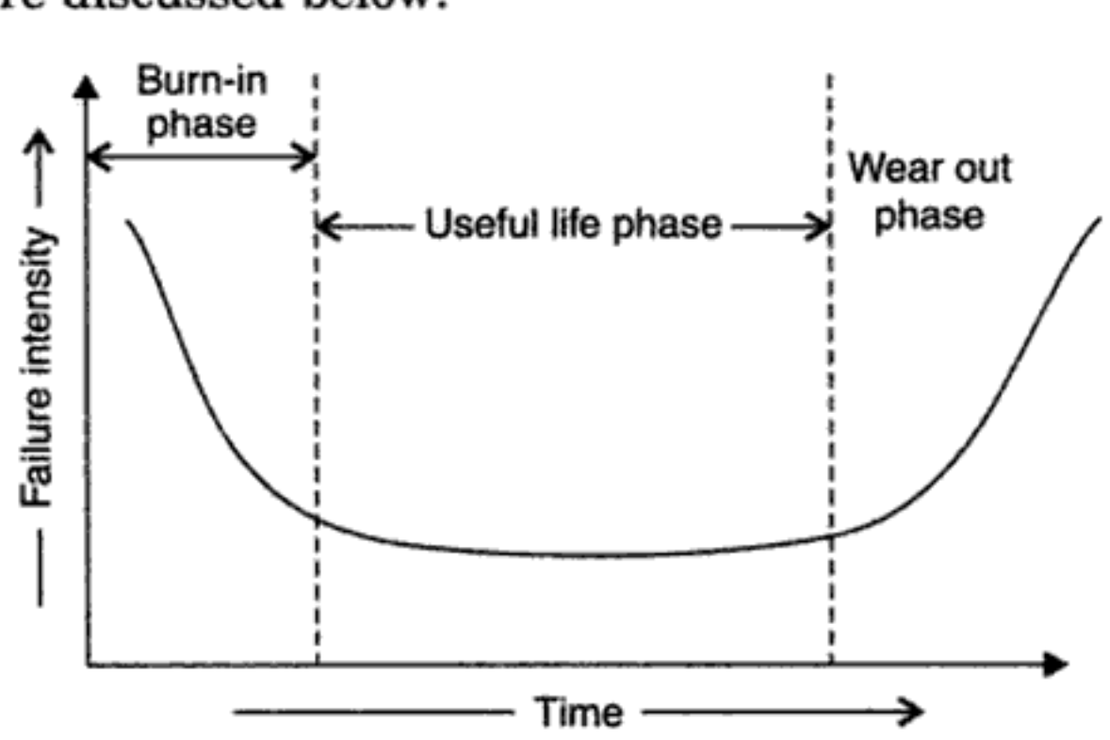


Fig. 1.5: Bath tub curve.

faults, failure intensity will come down initially and may stabilise after certain time. The second phase is the useful life phase where failure intensity is approximately constant and is called useful life of a product. After few years, again failure intensity will increase due to wearing out of components. This phase is called wear out phase. We do not have this phase for the software as it does not wear out. The curve for software is given in Fig. 1.6.

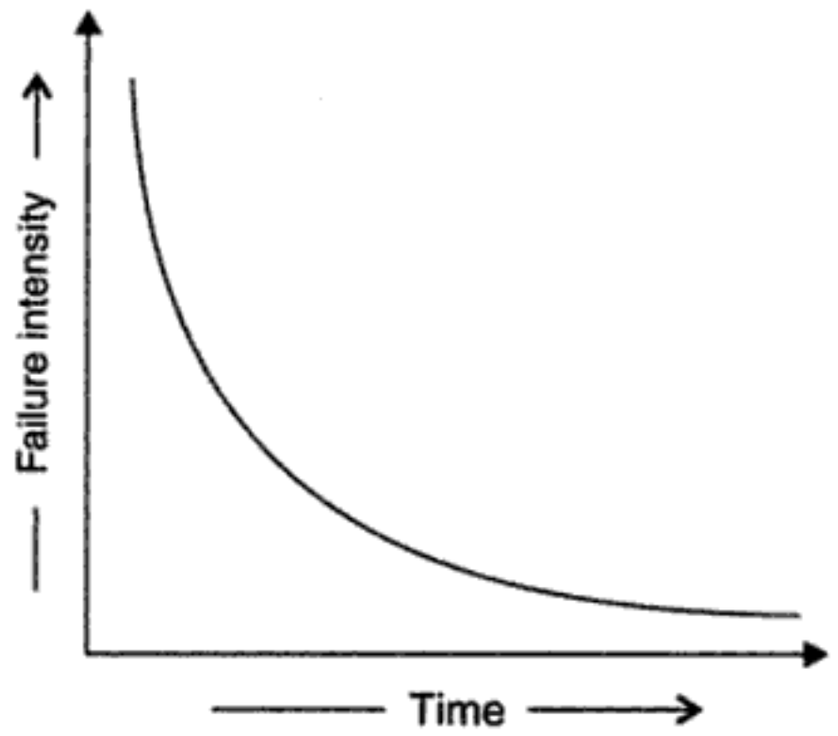


Fig. 1.6: Software curve.

Important point is software becomes reliable overtime instead of wearing out. It becomes obsolete, if the environment for which it was developed, changes. Hence software may be retired due to environmental changes, new requirements, new expectations, etc.

(*ii*) **Software is not manufactured.** The life of a software is from concept exploration to the retirement of the software product. It is one time development effort and continuous maintenance effort in order to keep it operational. However, making 1000 copies is not an issue and it does not involve any cost. In case of hardware product, every product costs us due to raw material and other processing expenses. We do not have assembly line in software development. Hence it is not manufactured in the classical sense.

(*iii*) **Reusability of components.** If we have to manufacture a TV, we may purchase picture tube from one vendor, cabinet from another, design card from third and other electronic components from fourth vendor. We will assemble every part and test the product thoroughly to produce a good quality TV. We may be required to manufacture only a few components or no component at all. We purchase every unit and component from the market and produce the finished product. We may have standard quality guidelines and effective processes to produce a good quality product.

In software, every project is a new project. We start from the scratch and design every unit of the software product. Huge effort is required to develop a software which further increases the cost of the software product. However, effort has been made to design standard components that may be used in new projects. Software reusability has introduced another area and is known as component based software engineering.

Hence developers can concentrate on truly innovative elements of design, that is, the parts of the design that represent something new. As explained earlier, in the hardware world, component reuse is a natural part of the engineering process. In software, there is only a humble beginning like graphical user interfaces are built using reusable components that

enable the creation of graphics windows, pull-down menus, and a wide variety of interaction mechanisms.

(*iv*) **Software is flexible.** We all feel that software is flexible. A program can be developed to do almost anything. Sometimes, this characteristic may be the best and may help us to accomodate any kind of change. However, most of the times, this "almost anything" characteristic has made software development difficult to plan, monitor and control. This unpredictability is the basis of what has been referred to for the past 35 years as the "Software Crisis".

1.2.5 Software Applications

Software has become integral part of most of the fields of human life. We name a field and we find the usage of software in that field. Software applications are grouped in to eight areas for convenience as shown in Fig. 1.7.

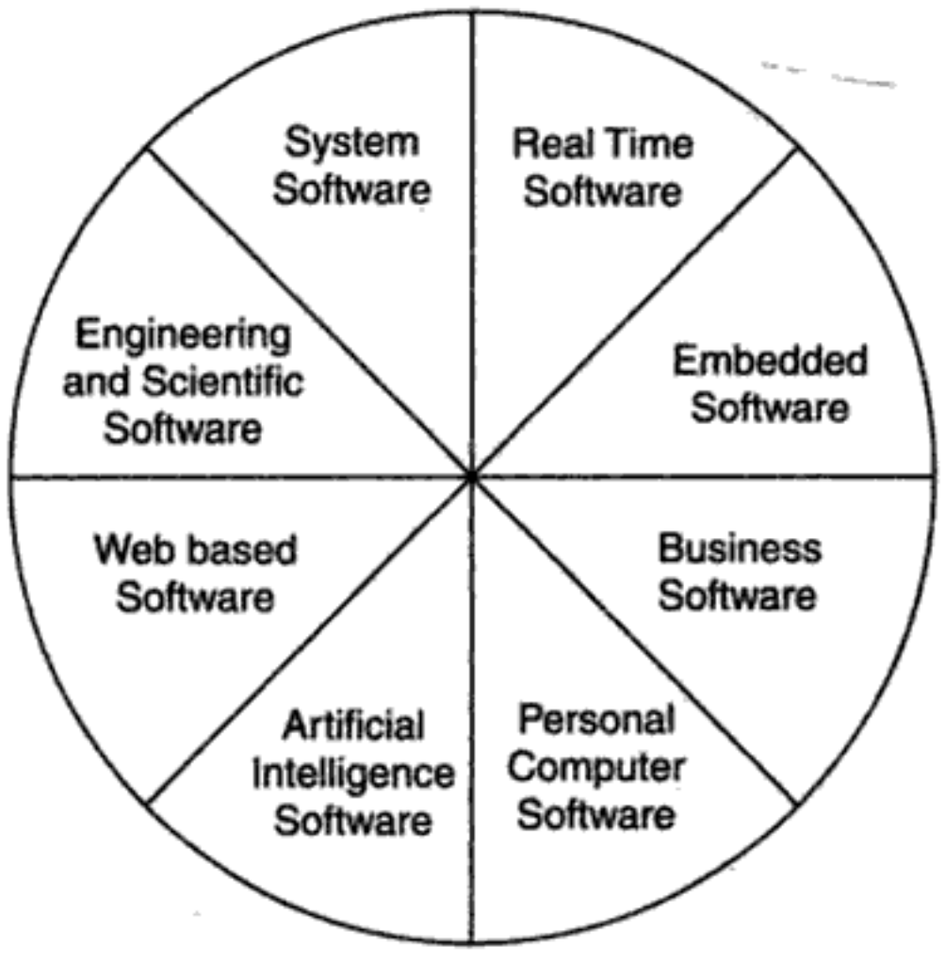


Fig. 1.7: Software applications.

(*i*) **System Software.** Infrastructure software come under this category like compilers, operating systems, editors, drivers, etc. Basically system software is a collection of programs to provide service to other programs.

(*ii*) **Real Time Software.** These software are used to monitor, control and analyze real world events as they occur. An example may be software required for weather forcasting. Such software will gather and process the status of temperature, humidity and other environmental parameters to forcast the weather.

(*iii*) **Embedded Software.** This type of software is placed in "Read-Only-Memory (ROM)" of the product and control the various functions of the product. The product could be an aircraft, automobile, security system, signalling system, control unit of power plants, etc. The embedded software handles hardware components and is also termed as intelligent software.

(*iv*) **Business Software.** This is the largest application area. The software designed to process business applications is called business software. Business software could be payroll, file monitoring system, employee management, account management. It may also be a data warehousing tool which helps us to take decisions based on available data. Management information system, enterprise resource planning (ERP) and such other software are popular examples of business software.

(*v*) **Personal Computer Software.** The software used in personal computers are covered in this category. Examples are word processors, computer graphics, multimedia and animating tools, database management, computer games etc. This is a very upcoming area and many big organisations are concentrating their effort here due to large customer base.

(*vi*) **Artificial Intelligence Software.** Artificial Intelligence Software makes use of non-numerical algorithms to solve complex problems that are not amenable to computation or straight forward analysis [PRESOI]. Examples are expert systems, artificial neural network, signal processing software etc.

(*vii*) **Web based Software.** The software related to web applications come under this category. Examples are CGI, HTML, Java, Perl, DHTML etc.

(*viii*) **Engineering and Scientific Software.** Scientific and engineering application software are grouped in this category. Huge computing is normally required to process data. Examples are CAD/CAM package, SPSS, MATLAB, Engineering Pro, Circuit analyzers etc.

1.3 SOME TERMINOLOGIES

Some terminologies are discussed in this section which are frequently used in the field of Software Engineering.

1.3.1 Deliverables and Milestones

Different deliverables are generated during software development. The examples are source code, user manuals, operating procedure manuals etc.

The milestones are the events that are used to ascertain the status of the project. Finalisation of specification is a milestone. Completion of design documentation is another milestone. The milestones are essential for project planning and management.

1.3.2 Product and Process

Product: What is delivered to the customer, is called a product. It may include source code, specification document, manuals, documentation etc. Basically, it is nothing but a set of deliverables only.

Process: Process is the way in which we produce software. It is the collection of activities that leads to (a part of) a product. An efficient process is required to produce good quality products.

If the process is weak, the end product will undoubtedly suffer, but an obsessive over-reliance on process is also dangerous.

1.3.3 Measures, Metrics and Measurement

The terms measures, metrics and measurement are often used interchangeably. It is interesting to understand the difference amongst these. A measure provides a quantitative indication of the extent, dimension, size, capacity, efficiency, productivity or reliability of some attributes of a product or process.

Measurement is the act of evaluating a measure. A metric is a quantitative measure of the degree to which a system, component, or process possesses a given attribute. Pressman [PRESO2] explained this very effectively with an example as given below:

"When a single data point has been collected (*e.g.*, the number of errors uncovered in the review of a single module), a measure has been established. Measurement occurs as the result of the collection of one or more data points (*e.g.*, a number of module reviews are investigated to collect measures of the number of errors in each module). A software metric relates the individual measures in some way (*e.g.*, the average number of errors found per review)."

Hence we collect measures and develop metrics to improve the software engineering practices.

1.3.4 Software Process and Product Metrics

Software metrics are used to quantitatively characterise different aspects of software process or software products. Process metrics quantify the attributes of software development process and environment; whereas product metrics are measures for the software product. Examples of process metrics include productivity, quality, failure rate, efficiency etc. Examples of product metrics are size, reliability, complexity, functionality etc.

1.3.5 Productivity and Effort

Productivity is defined as the rate of output, or production per unit of effort, *i.e.,* the output achieved with regard to the time taken but irrespective of the cost incurred. Hence, there are two issues for deciding the unit of measure

(*i*) quantity of output

(*ii*) period of time.

In software, one of the measure for quantity of output is lines of code (LOC) produced. Time is measured in days or months.

Hence most appropriate unit of effort is Person Months (PMs), meaning thereby number of persons involved for specified months. So, productivity may be measured as LOC/PM (lines of code produced/person month).

1.3.6 Module and Software Components

There are many definitions of the term module. They range from "a module is a FORTRAN subroutine" to "a module is an Ada Package", to "Procedures and functions of PASCAL and C", to "C++ Java classes" to "Java packages" to "a module is a work assignment for an individual developer". All these definitions are correct. The term subprogram is also used sometimes in place of module.

There are many definitions of software components. A general definition given by Alan W. Brown [BROW2K] is:

"An independently deliverable piece of functionality providing access to its services through interfaces".

Another definition from unified modeling language (UML) [OMG2K] is:

"A component represents a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces".

Hence, a reusable module is an independent and deliverable software part that encapsulates a functional specification and implementation for reuse by a third party.

However, a reusable component is an independent, deployable, and replaceable software unit that is reusable by a third party based on unit's specification, implementation, and well defined contracted interfaces.

1.3.7 Generic and Customised Software Products

The software products are divided in two categories:

(*i*) Generic products

(*ii*) Customised products.

Generic products are developed for anonymous customers. The target is generally the entire world and many copies are expected to be sold. Infrastructure software like operating systems, compilers, analysers, word processors, CASE tools etc. are covered in this category.

The customised products are developed for particular customers. The specific product is designed and developed as per customer requirements. Most of the development projects (say about 80%) come under this category.

1.4 ROLE OF MANAGEMENT IN SOFTWARE DEVELOPMENT

The management of software development is heavily dependent on four factors: People, Product, Process, and Project. Order of dependency is as shown in Fig. 1.8.

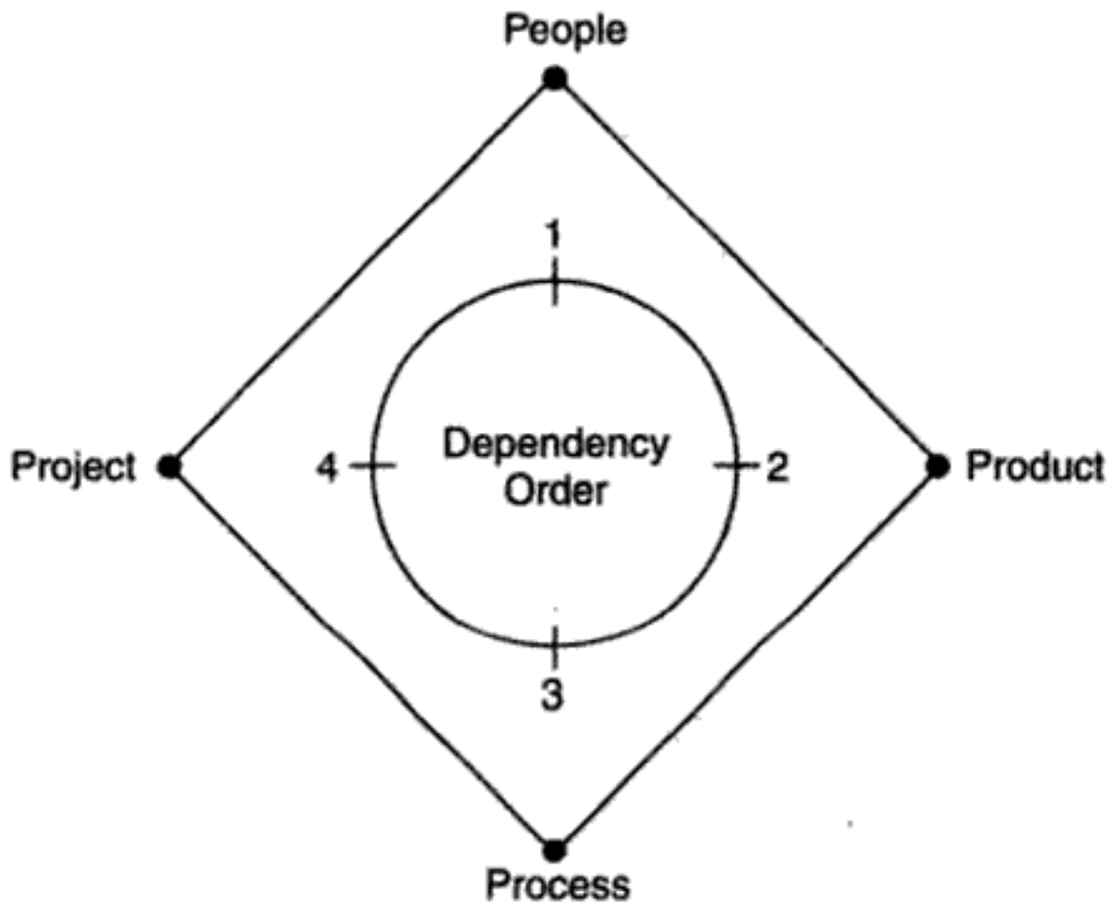


Fig. 1.8: Factors of management dependency (from People to Project).

Software development is a people centric activity. Hence, success of the project is on the shoulders of the people who are involved in the development.

1.4.1 The People

Software development requires good managers. The managers, who can understand the psychology of people and provide good leadership. A good manager can not ensure the success of the project, but can increase the probability of success. The areas to be given priority are: proper selection, training, compensation, career development, work culture etc.

Managers face challenges. It requires mental toughness to endure inner pain. We need to plan for the best, be prepared for the worst, expect surprises, but continue to move forward anyway. Charles Maurice once rightly said "I am more afraid of an army of one hundred sheep led by a lion than an army of one hundred lions led by a sheep".

Hence, manager selection is most crucial and critical. After having a good manager, project is in safe hands. It is the responsibility of a manager to manage, motivate, encourage, guide and control the people of his/her team.

1.4.2 The Product

What do we want to deliver to the customer? Obviously, a product; a solution to his/her problems.

Hence, objectives and scope of work should be defined clearly to understand the requirements. Alternate solutions should be discussed. It may help the managers to select a "best" approach within constraints imposed by delivery deadlines, budgetary restrictions, personnel availability, technical interfaces etc. Without well defined requirements, it may be impossible to define reasonable estimates of the cost, development time and schedule for the project.

1.4.3 The Process

The process is the way in which we produce software. It provides the framework from which a comprehensive plan for software development can be established. If the process is weak, the end product will undoubtedly suffer. There are many life cycle models and process improvements models. Depending on the type of project, a suitable model is to be selected. Now-a-days CMM (Capability Maturity Model) has become almost a standard for process framework. The process priority is after people and product, however, it plays very critical role for the success of the project. A small number of framework activities are applicable to all software projects, regardless of their size and complexity. A number of different task sets, tasks, milestones, work products, and quality assurance points, enable the framework activities to be adopted to the characteristics of the project and the requirements of the project team.

1.4.4 The Project

A proper planning is required to monitor the status of development and to control the complexity. Most of the projects are coming late with cost overruns of more than 100%. In order to manage a successful project, we must understand what can go wrong and how to do it right. We should define concrete requirements (although very difficult) and freeze these requirements. Changes should not be incorporated to avoid software surprises. Software surprises are always risky and we should minimise them. We should have a planning mechanism to give warning before the occurrence of any surprise.

All four factors (People, Product, Process and Project) are important for the success of the project. Their relative importance helps us to organise development activities in more scientific and professional way.

REFERENCES

[FRIT68] Bauer, Fritz et al., "Software Engineering: A Report on a Conference Sponsored by NATO Science Committee", NATO, 1968.

[BOEH89] Boehm B., "Risk Management", IEEE Computer Society Press, 1989.

[BROO87] Brooks F.P., "No Silver Bullet : Essence and Accidents of Software Engineering", IEEE Computer, 10—19, April, 1987.

[BROW2K] Brown A.W., "Large Scale, Component based Development", Englewood cliffs, NJ, PH, 2000.

[HUMP89] Humphrey W.S., "Managing the Software Process", Addison-Wesley Pub. Co., Reading, Massachusetts, USA, 1989.

[IBMG2K] IBM Global Services India Pvt. Ltd., Golden Tower, Airport Road, Bangalore, Letter of Bindu Subramani, Segment Manager—Corporate Training, March 9, 2000.

[LEVE95] Leveson N.G., "Software, System Safety and Computers", Addison Wesley, 1995.

[OMG2K] OMG Unified Modelling Language Specification, Version 1.4, Object Mangement Group, 2000.

[PRES02] Pressman R., "Software Engineering", McGraw Hill, 2002.

[PIER99] Piersal K., "Amusing Software Myths", www. bejeeber.org/Software-myths.html, 1999.

[SCHA90] Schach, Stephen, "Software Engineering", Vanderbilt University, Aksen Association, 1990.

[WIEG2K] Wiegers K.E., "Stop Promising Miracles" Software Development Magazine, February, 2000.

[WIEG94] Wiegers K.E., "Creating a Software Engineering Culture", Software Development Magazine, July, 1994.

[WIEG96] Wiegers K.E., "Software Process Improvement: Ten Traps to Avoid", Software Development Magazine, May, 1996.

[WIEG99] Wiegers K.E., "Why is Process Improvement So Hard", Software Development Magazine, February, 1999.

MULTIPLE CHOICE QUESTIONS

Note : Select most appropriate answer of the following questions.

1.1. Software is

(*a*) superset of programs
(*b*) subset of programs
(*c*) set of programs
(*d*) none of the above.

1.2. Which is NOT the part of operating procedure manuals?

(*a*) User manuals
(*b*) Operational manuals
(*c*) Documentation manuals
(*d*) Installation manuals.

1.3. Which is NOT a software characteristic?

(*a*) Software does not wear out
(*b*) Software is flexible
(*c*) Software is not manufactured
(*d*) Software is always correct.

1.4. Product is

(*a*) Deliverables
(*b*) User expectations
(*c*) Organisation's effort in development
(*d*) none of the above.

1.5. To produce a good quality product, process should be

(*a*) Complex
(*b*) Efficient
(*c*) Rigorous
(*d*) None of the above.

1.6. Which is not a product metric?

(*a*) Size
(*b*) Reliability
(*c*) Productivity
(*d*) Functionality.

1.7. Which is not a process metric?

(*a*) Productivity
(*b*) Functionality
(*c*) Quality
(*d*) Efficiency.

1.8. Effort is measured in terms of:

(*a*) Person-months (*b*) Rupees

(*c*) Persons (*d*) Months.

1.9. UML stands for

(*a*) Uniform modeling language (*b*) Unified modeling language

(*c*) Unit modeling language (*d*) Universal modeling language.

1.10. An independently deliverable piece of functionality providing access to its services through interfaces is called

(*a*) Software measurement (*b*) Software composition

(*c*) Software measure (*d*) Software component.

1.11. Infrastructure software are covered under

(*a*) Generic products (*b*) Customised products

(*c*) Generic and Customised products (*d*) none of the above.

1.12. Management of software development is dependent on

(*a*) people (*b*) product

(*c*) process (*d*) all of the above.

1.13. During software development, which factor is most crucial?

(*a*) People (*b*) Product

(*c*) Process (*d*) Project.

1.14. Program is

(*a*) subset of software (*b*) super set of software

(*c*) software (*d*) none of the above.

1.15. Milestones are used to

(*a*) know the cost of the project (*b*) know the status of the project

(*c*) know user expectations (*d*) none of the above.

1.16. The term module used during design phase refers to

(*a*) Function (*b*) Procedure

(*c*) Sub program (*d*) All of the above.

1.17. Software consists of

(*a*) Set of instructions + operating system

(*b*) Programs + documentation + operating procedures

(c) Programs + hardware manuals (*d*) Set of programs.

1.18. Software engineering approach is used to achieve:

(*a*) Better performance of hardware (*b*) Error free software

(*c*) Reusable software (*d*) Quality software product.

1.19. Concepts of software engineering are applicable to

(*a*) Fortran language only (*b*) Pascal language only

(*c*) 'C' language only (*d*) All of the above.

1.20. CASE Tool is

(*a*) Computer Aided Software Engineering (*b*) Component Aided Software Engineering

(*c*) Constructive Aided Software Engineering (*d*) Computer Analysis Software Engineering.

EXERCISES

1.1. Why is the primary goal of software development now shifting from producing good quality software to good quality maintainable software?

1.2. List the reasons for the "software crisis"? Why are CASE tools not normally able to control it?

1.3. "The software crisis is aggravated by the progress in hardware technology?" Explain with examples.

1.4. What is software crisis? Was Y2K a software crisis?

1.5. What is the significance of software crisis in reference to software engineering discipline.

1.6. How are software myths affecting software process? Explain with the help of examples.

1.7. State the difference between program and software. Why have documents and documentation become very important?

1.8. What is software engineering? Is it an art, craft or a science? Discuss.

1.9. What is the aim of software engineering? What does the discipline of software engineering discuss?

1.10. Define the term "Software Engineering". Explain the major differences between software engineering and other traditional engineering disciplines.

1.11. What is software process? Why is it difficult to improve it?

1.12. Describe the characteristics of software contrasting it with the characteristics of hardware.

1.13. Write down the major characteristics of a software. Illustrate with a diagram that the software does not wear out.

1.14. What are the components of a software? Discuss how a software differs from a program.

1.15. Discuss major areas of the applications of the software.

1.16. Is software a product or process? Justify your answer with examples.

1.17. Differentiate between the followings

(*i*) Deliverables and milestones (*ii*) Product and process

(*iii*) Measures, metrics and measurement

1.18. What is software metric? How is it different from software measurement?

1.19. Discuss software process and product metrics with the help of examples.

1.20. What is productivity? How is it related to effort? What is the unit of effort?

1.21. Differentiate between module and software component.

1.22. Distinguish between generic and customised software products. Which one has larger share of market and why?

1.23. Describe the role of management in software development with the help of examples.

1.24. What are various factors of management dependency in software development? Discuss each factor in detail.

1.25. What is more important: Product or process? Justify your answer.

2 Software Life Cycle Models

Contents

2 Software Life Cycle Models

The goal of software engineering is to provide models and processes that lead to the production of well-documented maintainable software in a manner that is predictable. For a mature process, it should be possible to determine in advance how much time and effort will be required to produce the final product. This can only be done using data from past experience, which requires that we must measure the software process.

Software development organizations follow some process when developing a software product. In immature organizations, the process is usually not written down. In mature organizations, the process is in writing and is actively managed. A key component of any software development process is the life cycle model on which the process is based. The particular life cycle model can significantly affect overall life cycle costs associated with a software product [RAKI97]. Life cycle of the software starts from concept exploration and ends at the retirement of the software.

In the IEEE standard Glossary of software Engineering Terminology, the software life cycle is:

"The period of time that starts when a software product is conceived and ends when the product is no longer available for use. The software life cycle typically includes a requirement phase, design phase, implementation phase, test phase, installation and check out phase, operation and maintenance phase, and sometimes retirement phase".

A software life cycle model is a particular abstraction that represents a software life cycle. A software life cycle model is often called a software development life cycle (SDLC).

2.1 *SDLC* MODELS

A variety of life cycle models have been proposed and are based on tasks involved in developing and maintaining software. Few well known life cycles models are discussed in this chapter.

2.1.1 Build and Fix Model

Sometimes a product is constructed without specifications or any attempt at design. Instead, the developer simply builds a product that is reworked as many times as necessary to satisfy the client [SCHA96].

This is an adhoc approach and not well defined. Basically, it is a simple two-phase model. The first phase is to write code and the next phase is to fix it as shown in Fig. 2.1. Fixing in this context may be error correction or addition of further functionality [TAKA96].

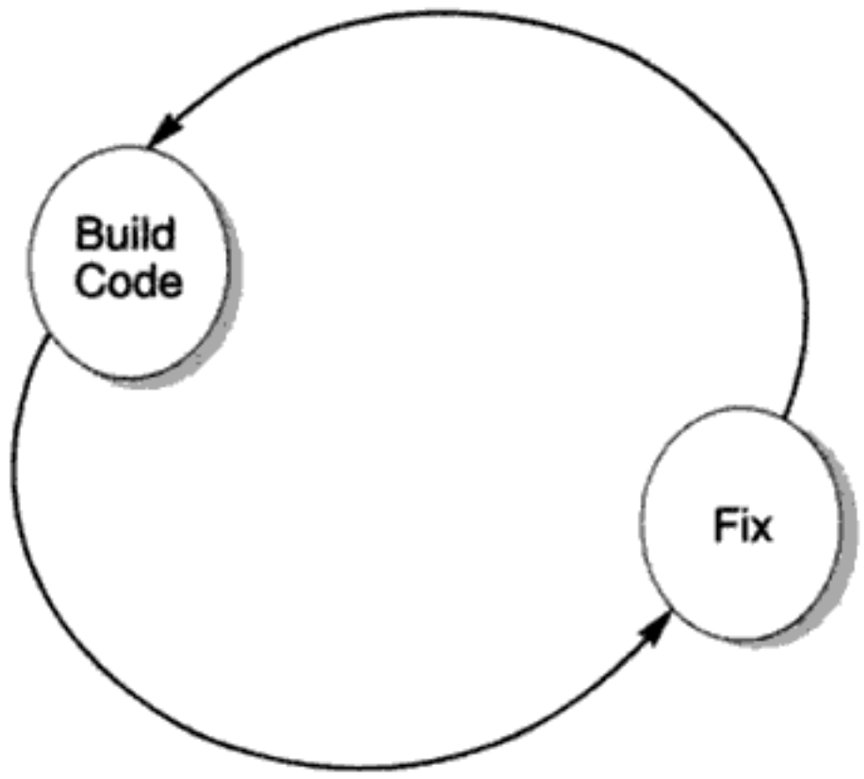


Fig. 2.1: Build and fix model.

Although this approach may work well on small programming exercises 100 or 200 lines long, this model is totally unsatisfactory for software of any reasonable size. Code soon becomes unfixable and unenhanceable. There is no room for design or any aspect of development process to be carried out in a structured or detailed way. The cost of the development using this approach is actually very high as compared to the cost of a properly specified and carefully designed product. In addition, maintenance of the product can be extremely difficult without specification or design documents.

2.1.2 The Waterfall Model

The most familiar model is the waterfall model, which is given in Fig. 2.2. This model has five phases: Requirements analysis and specification, design, implementation and unit testing, integration and system testing, and operation and maintenance. The phases always occur in this order and do not overlap. The developer must complete each phase before the next phase begins. This model is named "Waterfall Model", because its diagrammatic representation resembles a cascade of waterfalls.

1. **Requirement analysis and specification phase.** The goal of this phase is to understand the exact requirements of the customer and to document them properly. This activity is usually executed together with the customer, as the goal is to document all functions, performance and interfacing requirements for the software. The requirements describe the "what" of a system, not the "how". This phase produces a large document, written in a natural language, contains a description of what the system will do without describing how it will be done. The resultant document is known as software requirement specification (SRS) document.

The SRS document may act as contract between the developer and customer. If developer fails to implement full set of requirements, it may amount to failure to implement the contracted system.

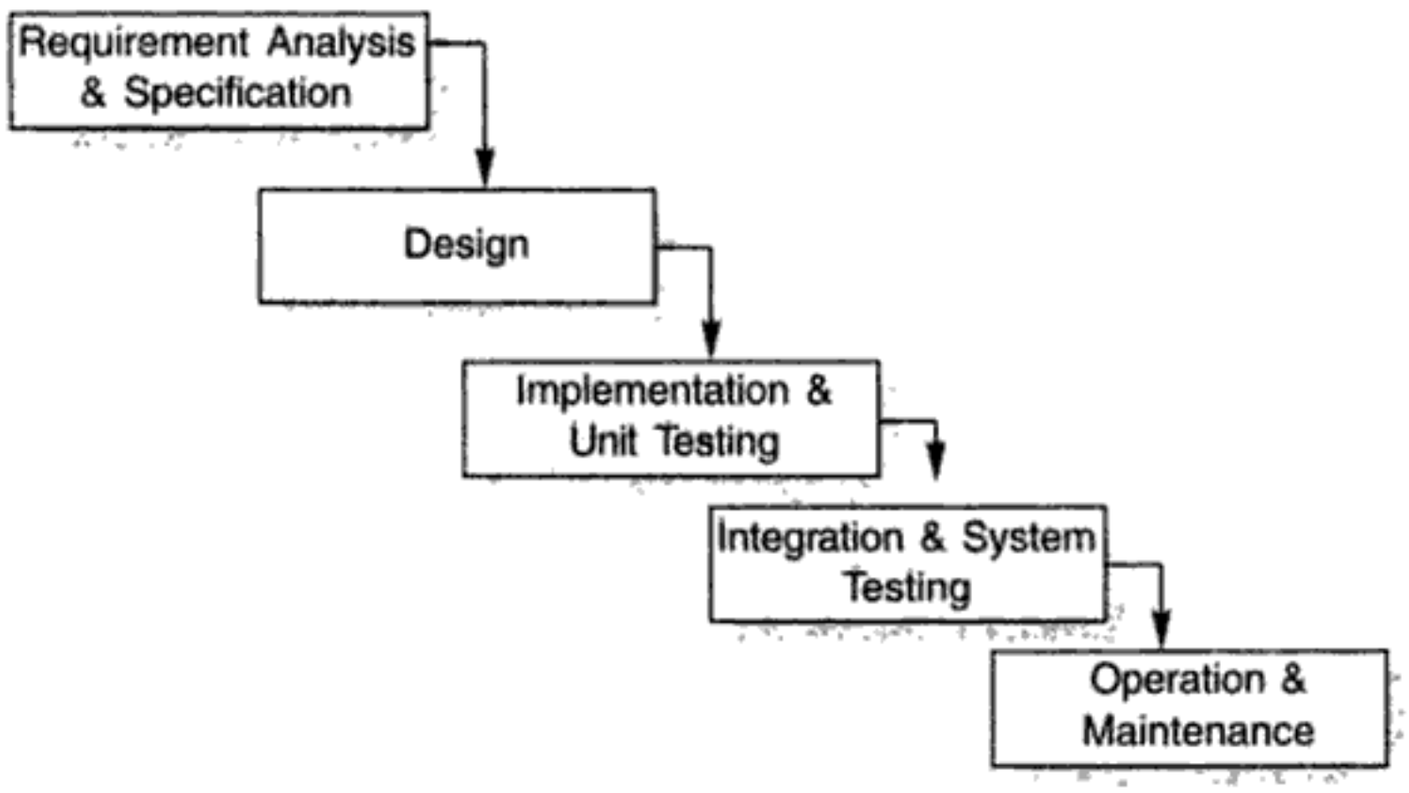


Fig. 2.2: Waterfall model

2. **Design phase.** The SRS document is produced in the previous phase, which contains the exact requirements of the customer. The goal of this phase is to transform the requirements specification into a structure that is suitable for implementation in some programming language. Here, overall software architecture is defined, and the high level and detailed design work is performed. This work is documented and known as software design description (SDD) document. The information contained in the SDD should be sufficient to begin the coding phase.

3. **Implementation and unit testing phase.** During this phase, design is implemented. If the SDD is complete, the implementation or coding phase proceeds smoothly, because all the information needed by the software developers is contained in the SDD.

During testing, the major activities are centered around the examination and modification of the code. Initially, small modules are tested in isolation from the rest of the software product. There are problems associated with testing a module in isolation. How do we run a module without anything to call it, to be called by it or, possibly, to output intermediate values obtained during execution? Such problems are solved in this phase and modules are tested after writing some overhead code.

4. **Integration and system testing phase.** This is a very important phase. Effective testing will contribute to the delivery of higher quality software products, more satisfied users, lower maintenance costs, and more accurate and reliable results. It is a very expensive activity and consumes one-third to one half of the cost of a typical development project.

As we know, the purpose of unit testing is to determine that each independent module is correctly implemented. This gives little chance to determine that the interface between modules is also correct, and for this reason integration testing is performed. System testing involves the testing of the entire system, whereas software is a part of the system. This is essential to build confidence in the developers before software is delivered to the customer or released in the market.

5. **Operation and maintenance phase.** Software maintenance is a task that every development group has to face, when the software is delivered to the customer's site, installed and is operational. Therefore, release of software inaugurates the operation and maintenance phase of the life cycle. The time spent and effort required to keep the software operational

after release is very significant. Despite the fact that it is a very important and challenging task; it is routinely the poorly managed headache that nobody wants to face.

Software maintenance is a very broad activity that includes error correction, enhancement of capabilities, deletion of obsolete capabilities, and optimization. The purpose of this phase is to preserve the value of the software over time. This phase may span for 5 to 50 years whereas development may be 1 to 3 years.

This model is easy to understand and reinforces the notion of "define before design" and "design before code". This model expects complete and accurate requirements early in the process, which is unrealistic. Working software is not available until relatively late in the process, thus delaying the discovery of serious errors. It also does not incorporate any kind of risk assessment.

Problems of waterfall model

(*i*) It is difficult to define all requirements at the beginning of a project.

(*ii*) This model is not suitable for accomodating any change.

(*iii*) A working version of the system is not seen until late in the project's life.

(*iv*) It does not scale up well to large projects.

(*v*) Real projects are rarely sequential.

Due to these weaknesses, the application of waterfall model should be limited to situations where the requirements and their implementation are well understood. For example, if an organisation has experience in developing accounting systems then building a new accounting system based on existing designs could be easily managed with the waterfall model.

2.1.3 Prototyping Model

A disadvantage of waterfall model as discussed in the last section is that the working software is not available until late in the process, thus delaying the discovery of serious errors. An alternative to this is to first develop a working prototype of the software instead of developing the actual software. The working prototype is developed as per current available requirements. Basically, it has limited functional capabilities, low reliability, and untested performance (usually low).

The developers use this prototype to refine the requirements and prepare the final specification document. Because the working prototype has been evaluated by the customer, it is reasonable to expect that the resulting specification document will be correct. When the prototype is created, it is reviewed by the customer. Typically this review gives feedback to the developers that helps to remove uncertainties in the requirements of the software, and starts an iteration of refinement in order to further clarify requirements as shown in Fig. 2.3.

The prototype may be a usable program, but is not suitable as the final software product. The reason may be poor performance, maintainability or overall quality. The code for the prototype is thrown away; however the experience gathered from developing the prototype helps in developing the actual system. Therefore, the development of a prototype might involve extra cost, but overall cost might turnout to be lower than that of an equivalent system developed using the waterfall model.

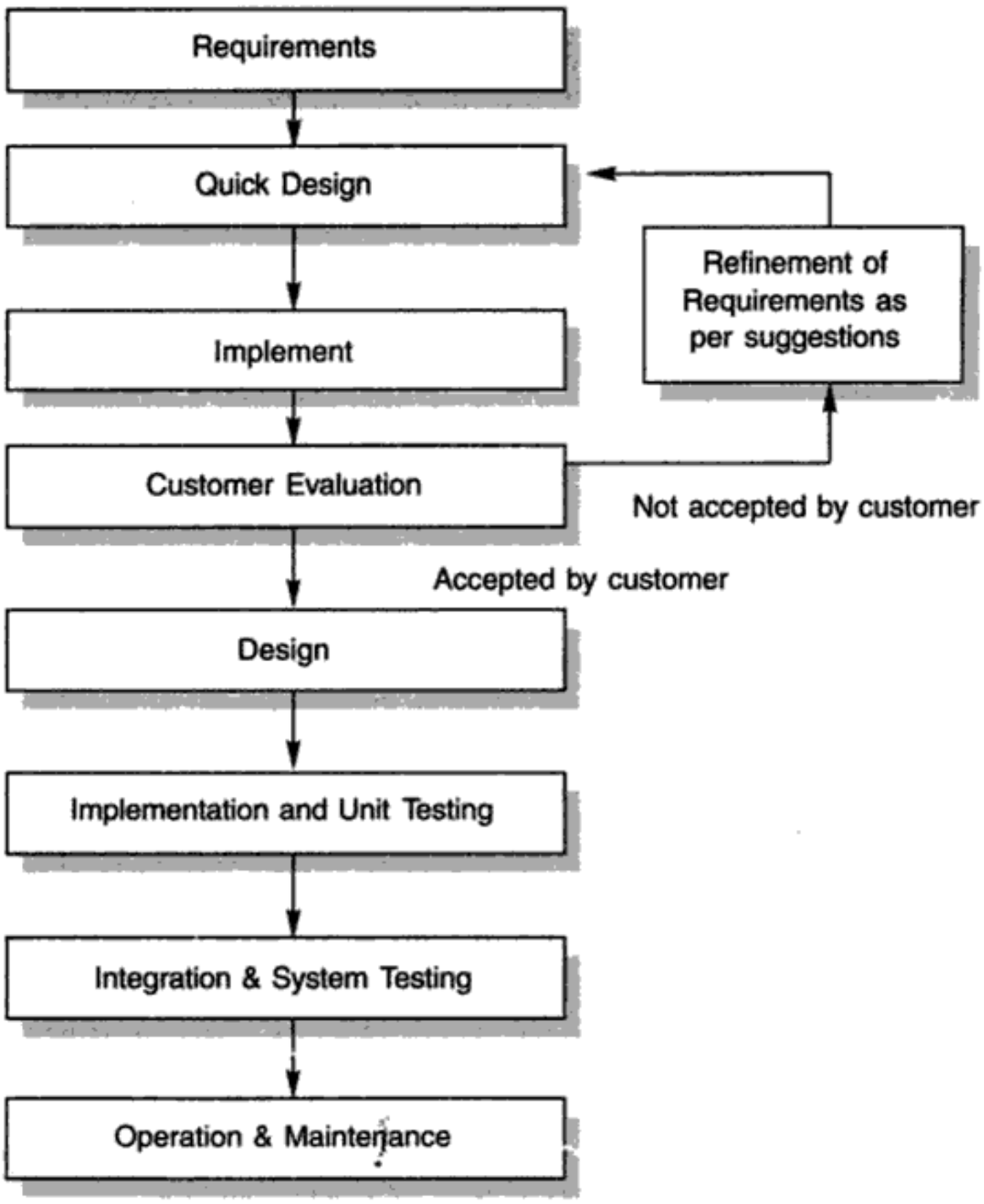


Fig. 2.3: Prototyping model

The developers should develop prototype as early as possible to speed up the software development process. After all, the sole use of this is to determine the customer's real needs. Once this has been determined, the prototype is discarded. For this reason, the internal structure of the prototype is not very important [SCHA96].

After the finalization of software requirement and specification (SRS) document, the prototype is discarded and actual system is then developed using the waterfall approach. Thus, it is used as an input to waterfall model and produces maintainable and good quality software. This model requires extensive participation and involvement of the customer, which is not always possible.

2.1.4 Iterative Enhancement Model

This model has the same phases as the waterfall model, but with fewer restrictions. Generally the phases occur in the same order as in the waterfall model, but these may be conducted in several cycles. A useable product is released at the end of the each cycle, with each release providing additional functionality [BASI75].

During the first requirements analysis phase, customers and developers specify as many requirements as possible and prepare a SRS document. Developers and customers then prioritize

these requirements. Developers implement the specified requirements in one or more cycles of design, implementation and test based on the defined priorities. The model is given in Fig. 2.4.

The aim of the waterfall and prototyping models is the delivery of a complete, operational and good quality product. In contrast, this model does deliver an operational quality product at each release, but one that satisfies only a subset of the customer's requirements. The complete product is divided into releases, and the developer delivers the product release by release. A typical product will usually have many releases as shown in Fig. 2.4. At each release, customer has an operational quality product that does a portion of what is required. The customer is able to do some useful work after first release. With this model, first release may be available within few weeks or months, whereas the customer generally waits months or years to receive a product using the waterfall and prototyping model.

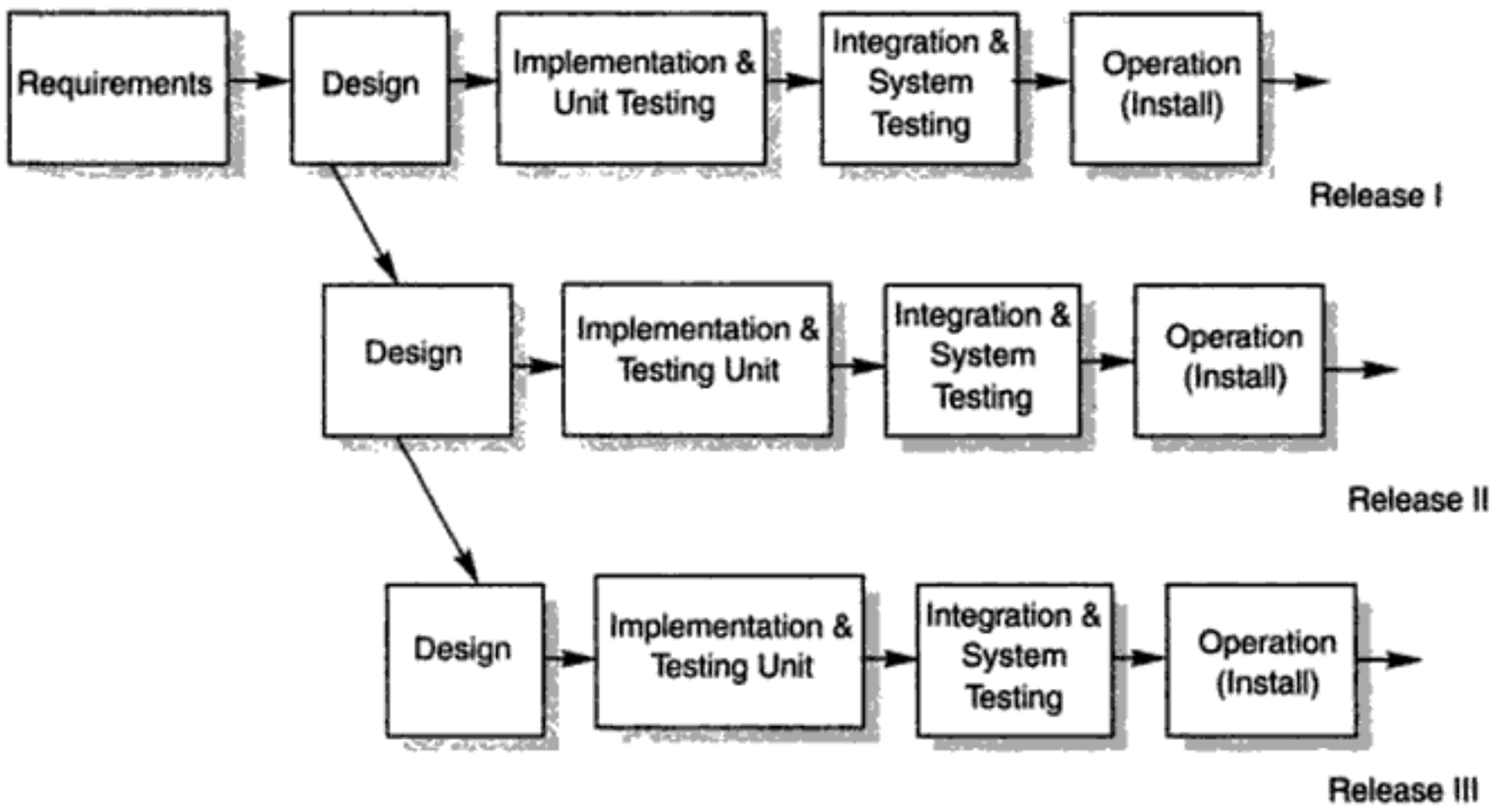


Fig. 2.4: Iterative enhancement model.

2.1.5 Evolutionary Development Model

Evolutionary development model resembles iterative enhancement model. The same phases as defined for the waterfall model occur here in a cyclical fashion. This model differs from iterative enhancement model in the sense that this does not require a useable product at the end of each cycle. In evolutionary development, requirements are implemented by category rather than by priority.

For example, in a simple database application, one cycle might implement the graphical user interface (GUI); another file manipulation; another queries; and another updates. All four cycles must complete before there is working product available. GUI allows the users to interact with the system; file manipulation allows data to be saved and retrieved; queries allow users to get data out of the system; and updates allow users to put data into the system. With any one of those parts missing, the system would be unusable.

In contrast, an iterative enhancement model would start by developing a very simplistic, but usable database. On the completion of each cycle, the system would become more sophisticated. It, would, however, provide all the critical functionality by the end of the first

cycle. Evolutionary development and iterative enhancement are somewhat interchangeable. Evolutionary development should be used when it is not necessary to provide a minimal version of the system quickly.

This model is useful for projects using new technology that is not well understood. This is also used for complex projects where all functionality must be delivered at one time, but the requirements are unstable or not well understood at the beginning.

2.1.6 Spiral Model

The problem with traditional software process models is that they do not deal sufficiently with the uncertainty, which is inherent to software projects. Important software projects have failed because project risks were neglected and nobody was prepared when something unforeseen happened. Barry Boehm recognized this and tried to incorporate the "project risk" factor into a life cycle model. The result is the spiral model, which was presented in 1986 [BOEH86] and is shown in Fig. 2.5.

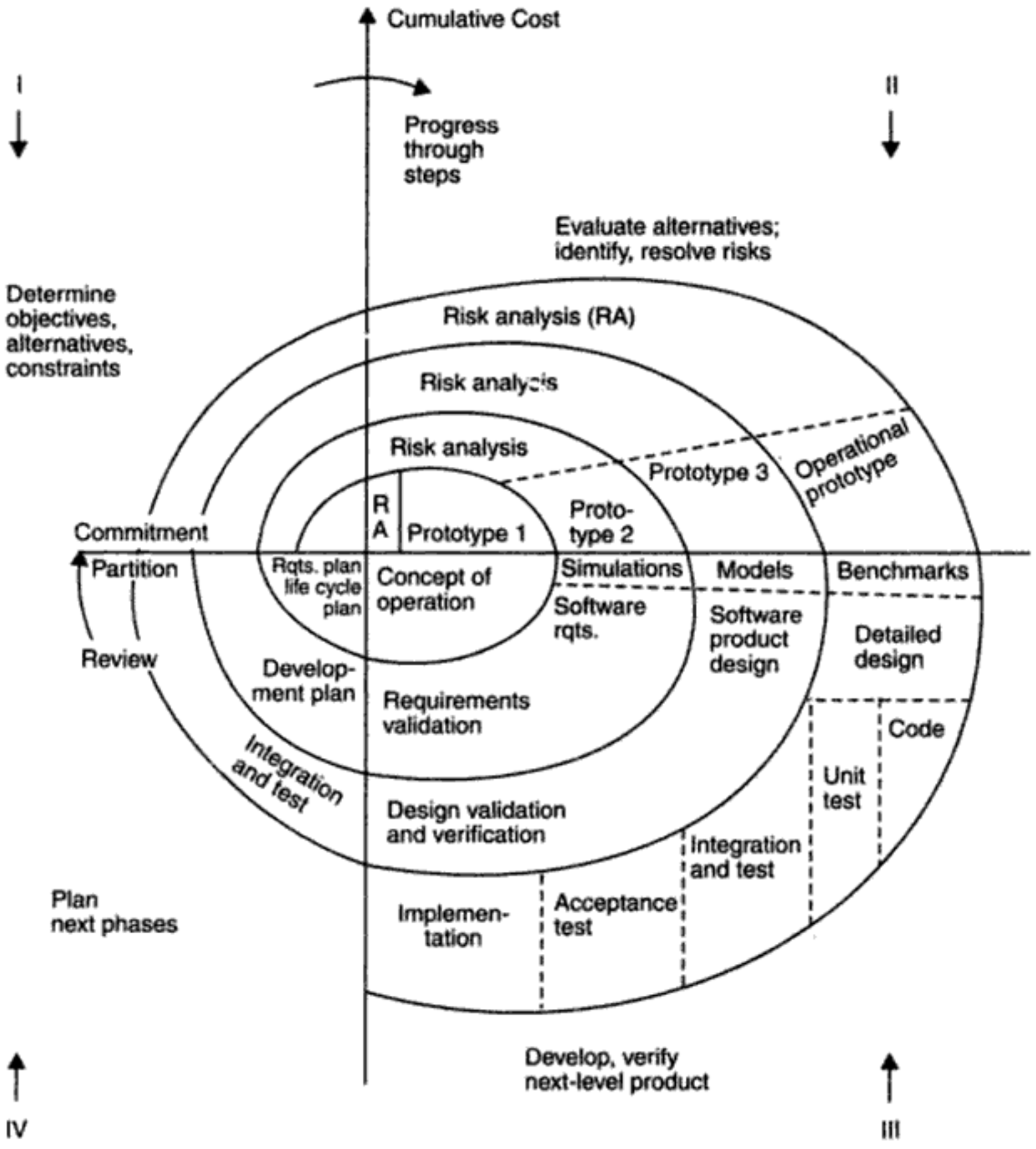


Fig. 2.5: Spiral model

The radial dimension of the model represents the cumulative costs. Each path around the spiral is indicative of increased costs. The angular dimension represents the progress made

in completing each cycle. Each loop of the spiral from X-axis clockwise through 360° represents one phase. One phase is split roughly into four sectors of major activities:

- Planning: Determination of objectives, alternatives and constraints
- Risk Analysis: Analyze alternatives and attempts to identify and resolve the risks involved
- Development: Product development and testing product
- Assessment: Customer evaluation

During the first phase, planning is performed, risks are analyzed, prototypes are built, and customers evaluate the prototype. During the second phase, a more refined prototype is built, requirements are documented and validated, and customers are involved in assessing the new prototype. By the time third phase begins, risks are known, and a somewhat more traditional development approach is taken [RAKI97].

The focus is the identification of problems and the classification of these into different levels of risks, the aim being to eliminate high-risk problems before they threaten the software operation or cost.

An important feature of the spiral model is that each phase is completed with a review by the people concerned with the project (designers and programmers). This review consists of a review of all the products developed up to that point and includes the plans for the next cycle. These plans may include a partition of the product in smaller portions for development or components that are implemented by individual groups or persons. If the plan for the development fails, then the spiral is terminated. Otherwise, it terminates with the initiation of new or modified software.

The advantage of this model is the wide range of options to accommodate the good features of other life cycle models. It becomes equivalent to another life cycle model in appropriate situations. It also incorporates software quality objectives into software development. The risk analysis and validation steps eliminate errors in the early phases of development.

The spiral model has some difficulties that need to be resolved before it can be a universally applied life cycle model. These difficulties include lack of explicit process guidance in determining objectives, constraints, alternatives; relying on risk assessment expertise; and providing more flexibility than required for many applications.

2.1.7 The Rapid Application Development (RAD) Model

This model was proposed by IBM in the 1980s through the book of James Martin entitled "Rapid Application Development". Here, user involvement is essential from requirement phase to delivery of the product. The continuous user participation ensures the involvement of user's expectations and perspective in requirements elicitation, analysis and design of the system.

The process is started with building a rapid prototype and is given to user for evaluation. The user feedback is obtained and prototype is refined. The process continues, till the requirements are finalised. We may use any grouping technique (like FAST, QFD, Brainstorming Sessions; for details refer chapter 3) for requirements elicitation. Software requirement and specification (SRS) and design documents are prepared with the association of users.

There are four phases in this model and these are shown in Fig. 2.6.

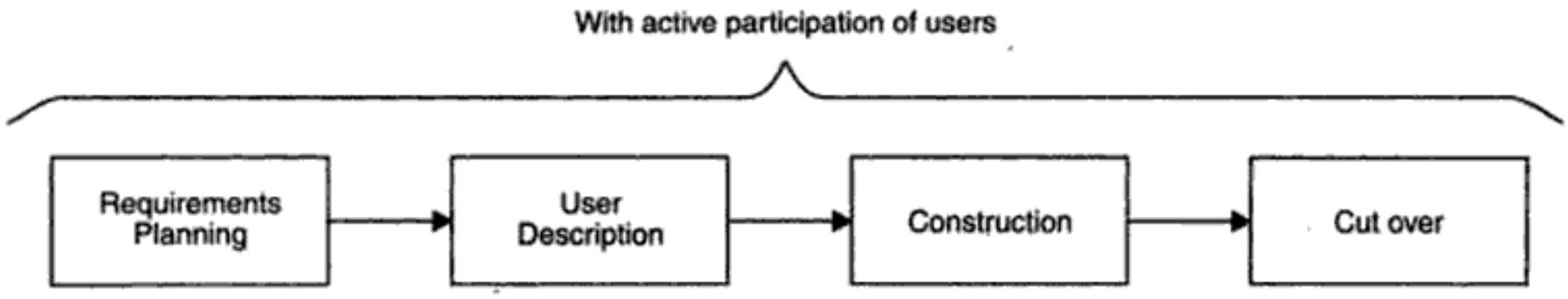


Fig. 2.6: RAD Model

(*i*) **Requirements planning phase.** Requirements are captured using any group elicitation technique. Some techniques are discussed in chapter 3. Only issue is the active involvement of users for understanding the project.

(*ii*) **User Description.** Joint teams of developers and users are constituted to prepare, understand and review the requirements. The team may use automated tools to capture information from the other users.

(*iii*) **Construction phase.** This phase combines the detailed design, coding and testing phase of waterfall model. Here, we release the product to customer. It is expected to use code generators, screen generators and other types of productivity tools.

(*iv*) **Cut over phase.** This phase incorporates acceptance testing by the users, installation of the system, and user training.

In this model, quick initial views about the product are possible due to delivery of rapid prototype. The development time of the product may be reduced due to use of powerful development tools. It may use CASE tools and frameworks to increase productivity. Involvement of user may increase the acceptability of the product.

If user cannot be involved throughout the life cycle, this may not be an appropriate model. Development time may not be reduced very significantly, if reusable components are not available. Highly specialized and skilled developers are expected and such developers may not be available very easily. It may not be effective, if system can not be properly modularised.

2.2 SELECTION OF A LIFE CYCLE MODEL

The selection of a suitable model is based on the following characteristics/categories:

(*i*) Requirements

(*ii*) Development team

(*iii*) Users

(*iv*) Project type and associated risk.

2.2.1 Characteristics of Requirements

Requirements are very important for the selection of an appropriate model. There are number of situations and problems during requirements capturing and analysis. The details are given in Table 2.1.

Table 2.1: Selection of a model based on characteristics of requirements

Requirements	*Waterfall*	*Prototype*	*Iterative enhancement*	*Evolutionary development*	*Spiral*	*RAD*
Are requirements easily understandable and defined?	Yes	No	No	No	No	Yes
Do we change requirements quite often?	No	Yes	No	No	Yes	No
Can we define requirements early in the cycle?	Yes	No	Yes	Yes	No	Yes
Requirements are indicating a complex system to be built	No	Yes	Yes	Yes	Yes	No

2.2.2 Status of Development Team

The status of development team in terms of availability, effectiveness, knowledge, intelligence, team work etc., is very important for the success of the project. If we know above mentioned parameters and characteristics of the team, then we may choose an appropriate life cycle model for the project. Some of the details are given in Table 2.2.

Table 2.2: Selection based on status of development team

Development team	*Waterfall*	*Prototype*	*Iterative enhancement*	*Evolutionary development*	*Spiral*	*RAD*
Less experience on similar projects	No	Yes	No	No	Yes	No
Less domain knowledge (new to the technology)	Yes	No	Yes	Yes	Yes	No
Less experience on tools to be used	Yes	No	No	No	Yes	No
Availability of training, if required	No	No	Yes	Yes	No	Yes

2.2.3 Involvement of Users

Involvement of users increases the understandability of the project. Hence user participation, if available, plays a very significant role in the selection of an appropriate life cycle model. Some issues are discussed in Table 2.3.

Table 2.3: Selection based on user's participation

Involvement of Users	*Waterfall*	*Prototype*	*Iterative enhancement*	*Evolutionary development*	*Spiral*	*RAD*
User involvement in all phases	No	Yes	No	No	No	Yes
Limited user participation	Yes	No	Yes	Yes	Yes	No
User have no previous experience of participation in similar projects	No	Yes	Yes	Yes	Yes	No
Users are experts of problem domain	No	Yes	Yes	Yes	No	Yes

2.3.4 Type of Project and Associated Risk

Very few models incorporate risk assessment. Project type is also important for the selection of a model. Some issues are discussed in Table 2.4.

Table 2.4: Selection based on type of project with associated risk

Project type and risk	*Waterfall*	*Prototype*	*Iterative enhancement*	*Evolutionary development*	*Spiral*	*RAD*
Project is the enhancement of the existing system	No	No	Yes	Yes	No	Yes
Funding is stable for the project	Yes	Yes	No	No	No	Yes
High reliability requirements	No	No	Yes	Yes	Yes	No
Tight project schedule	No	Yes	Yes	Yes	Yes	Yes
Use of reusable components	No	Yes	No	No	Yes	Yes
Are resources (time, money people etc.) scarce?	No	Yes	No	No	Yes	No

An appropriate model may be selected based on options given in four Tables (*i.e.*, Table 2.1 to 2.4). Firstly, we have to answer the questions presented for each category by circling a yes or no in each table. Rank the importance of each category, or question within the category, in terms of the project for which we want to select a model. The total number of circled responses for each column in the tables decide an appropriate model. We may also use the category ranking to resolve the conflicts between models if the total in either case is close or the same.

REFERENCES

[BASI75] Basili V.R., "Iterative Enhancement: A Practical Technique for Software Development", IEEE Trans on Software Engineering, SE–1, No. 4, 390–396, December 1975.

[BOEH86] Boehm B., "A Spiral Model for Software Development and Enhancement", ACM Software Engineering Notes, 14–24, August, 1986.

[RAKI97] Rakitin S.R., "Software Verification and Validation", Artech House Inc., Norwood, MA, 1997.

[SCHA96] Schach S., "Classical and Object Oriented Software Engineering", IRWIN, USA, 1996.

[TAKA96] Takang A.A. and P.A. Grubb, "Software Maintenance–Concepts and Practice", Int. Thomson Computer Press, Cambridge, U.K., 1996.

MULTIPLE CHOICE QUESTIONS

Note: Choose most appropriate answer of the following questions.

2.1. Spiral Model was developed by

(*a*) Bev Littlewood (*b*) Berry Boehm

(*c*) Roger Pressman (*d*) Victor Basili.

2.2. Which model is most popular for student's small projects?

(*a*) Waterfall model (*b*) Spiral model

(*c*) Quick and fix model (*d*) Prototyping model.

2.3. Which is not a software life cycle model?

(*a*) Waterfall model (*b*) Spiral model

(*c*) Prototyping model (*d*) Capability maturity model.

2.4. Project risk factor is considered in

(*a*) Waterfall model (*b*) Prototyping model

(*c*) Spiral model (*d*) Iterative enhancement model.

2.5. SDLC stands for

(*a*) Software design life cycle (*b*) Software development life cycle

(*c*) System development life cycle (*d*) System design life cycle.

2.6. Build and fix model has

(*a*) 3 phases (*b*) 1 phase

(*c*) 2 phases (*d*) 4 phases.

2.7. SRS Stands for

(*a*) Software requirements specification (*b*) Software requirements solutions

(*c*) System requirements specification (*d*) none of the above.

2.8. Waterfall model is not suitable for

(*a*) small projects (*b*) accomodating change

(*c*) complex projects (*d*) none of the above.

2.9. RAD stands for

(*a*) Rapid application development (*b*) Relative application development

(*c*) Ready application development (*d*) Repeated application development.

2.10. RAD model was proposed by
(*a*) Lucent Technologies (*b*) Motorola
(*c*) IBM (*d*) Microsoft.

2.11. If requirements are easily understandable and defined, which model is best suited?
(*a*) Waterfall model (*b*) Prototyping model
(*c*) Spiral model (*d*) None of the above.

2.12. If requirements are frequently changing, which model is to be selected
(*a*) Waterfall (*b*) Prototyping model
(*c*) RAD model (*d*) Iterative enhancement model.

2.13. If user participation is available, which model is to be chosen?
(*a*) Waterfall model (*b*) Iterative enhancement model
(*c*) Spiral model (*d*) RAD model.

2.14. If limited user participation is available, which model is to be selected?
(*a*) Waterfall model (*b*) Spiral model
(*c*) Iterative enhancement model (*d*) any of the above.

2.15. If project is the enhancement of existing system, which model is best suited?
(*a*) waterfall model (*b*) Prototyping model
(*c*) Iterative enhancement model (*d*) spiral model.

2.16. Which one is the most important feature of spiral model?
(*a*) Quality management (*b*) Risk management
(*c*) Performance management (*d*) Efficiency management.

2.17. Most suitable model for new technology that is not well understood is:
(*a*) Waterfall model (*b*) RAD model
(c) Iterative enhancement model (*d*) Evolutionary development model.

2.18. Statistically, the maximum percentage of errors belong to the following phase of SDLC
(*a*) Coding (*b*) Design
(*c*) Specifications (*d*) Installation and maintenance.

2.19. Which phase is not available in software life cycle?
(*a*) Coding (*b*) Testing
(*c*) Maintenance (*d*) Abstraction.

2.20. The development is supposed to proceed linearly through the phases in
(*a*) Spiral model (*b*) Waterfall model
(*c*) Prototyping model (*d*) None of the above.

EXERCISES

2.1. What do you understand by the term Software Development Life Cycle (SDLC)? Why is it important to adhere to a life cycle model while developing a large software product?

2.2. What is software life cycle? Discuss the generic waterfall model.

2.3. List the advantages of using waterfall model instead of adhoc build and fix model.

2.4. Discuss the prototype model. What is the effect of designing a prototype on the overall cost of the software project?

2.5. What are the advantages of developing the prototype of a system?

2.6. Describe the type of situations where iterative enhancement model might lead to difficulties.

2.7. Compare iterative enhancement model and evolutionary development model.

2.8. Sketch a neat diagram of spiral model of software life cycle.

2.9. Compare the waterfall model and the spiral model of software development.

2.10. As we move outward along with process flow path of the spiral model, what can we say about the software that is being developed or maintained?

2.11. How does "project risk" factor affect the spiral model of software development?

2.12. List the advantages and disadvantages of involving a software engineer throughout the software development planning process.

2.13. Explain the spiral model of software development. What are the limitations of such a model?

2.14. Describe the rapid application development (RAD) model. Discuss each phase in detail.

2.15. What are the characteristics to be considered for the selection of a life cycle model?

2.16. What is the role of user participation in the selection of a life cycle model?

2.17. Why do we feel that characteristics of requirements play a very significant role in the selection of a life cycle model?

2.18. Write short note on "status of development team" for the selection of a life cycle model.

2.19. Discuss the selection process parameters for a life cycle model.

3 Software Requirements Analysis and Specifications

Contents

3 Software Requirements Analysis and Specifications

When we receive a request for a new software project from the customer, first of all, we would like to understand the project. The new project may replace the existing system such as preparation of students semester results electronically rather than manually. Sometimes, the new project is an enhancement or extension of a current (manual or automated) system. For example, a web enabled student result declaration system that would enhance the capabilities of the current result declaration system. No matter, whether its functionality is old or new, each project has a purpose, usually expressed in what the system can do. Hence, goal is to understand the requirements of the customer and document them properly. A requirement is a feature of the system or a description of something the system is capable of doing in order to fulfil the system's purpose.

The hardest part of building a software system is deciding precisely what to build. No other part of the conceptual work is so difficult as establishing the detailed technical requirements. No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later [BR0095]. Throughout software industry's history, we have struggled with this truth. Defining and applying good, complete requirements is hard to work, and success in this endeavor has eluded many of us. Yet, we continue to make progress.

3.1 REQUIREMENTS ENGINEERING

Requirements describe the "what" of a system, not the "how". Requirements engineering produces one large document, written in a natural language, contains a description of what the system will do without describing how it will do. The input to requirements engineering is the problem statement prepared by the customer. The problem statement may give an overview of the existing system alongwith broad expections from the new system

3.1.1 Crucial Process Steps

The quality of a software product is only as good as the process that creates it. Requirements engineering is one of the most crucial activity in this creation process. Without well-written requirements specifications, developers do not know what to build, customers do not know what to expect, and there is no way to validate that the built system satisfies the requirements.

Requirements engineering is the disciplined application of proven principles, methods, tools, and notations to describe a proposed system's intended behaviour and its associated constraints [HSIA93]. This process consists of four steps as shown in Fig. 3.1.

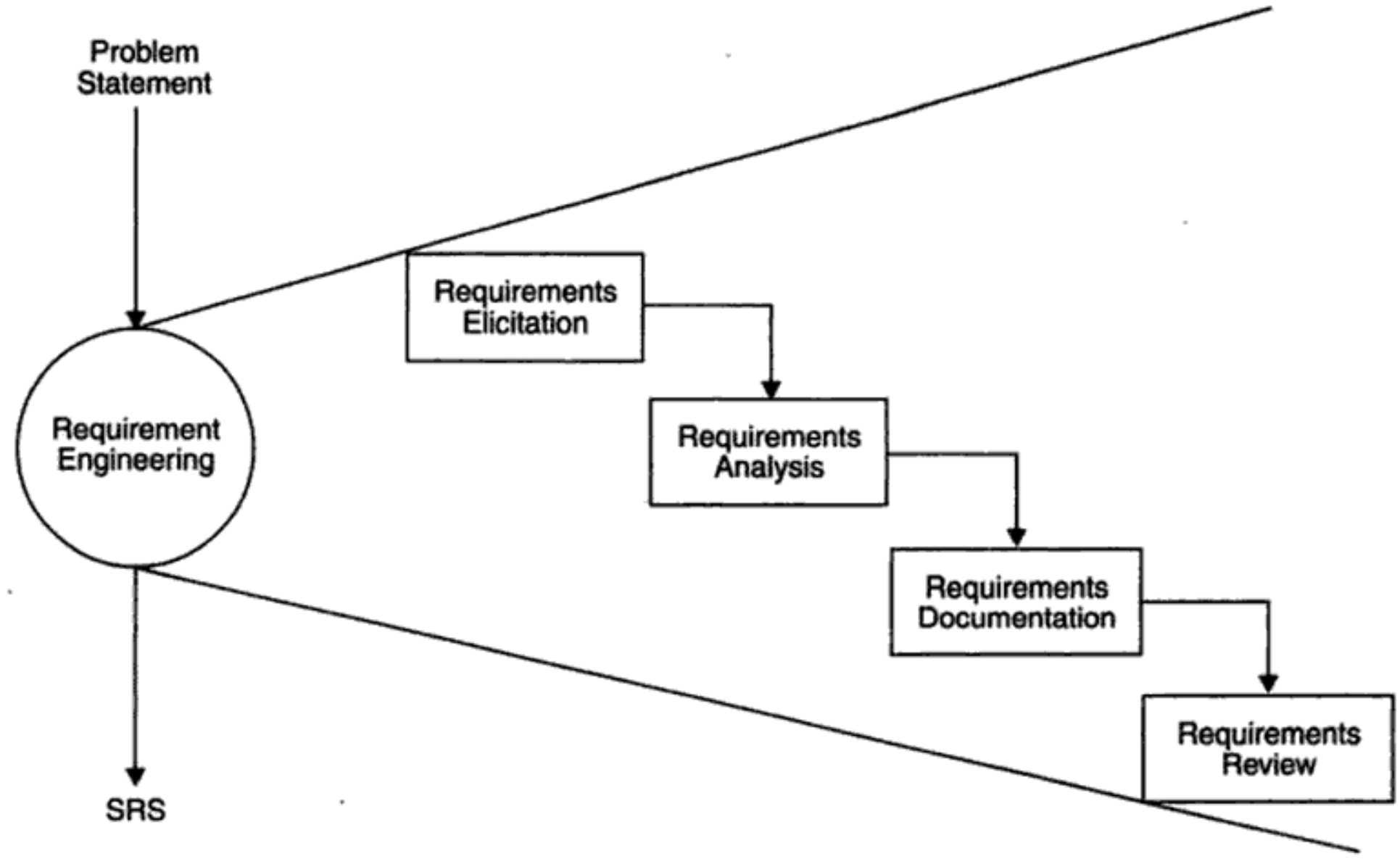


Fig. 3.1: Crucial process steps of requirement engineering.

(*i*) **Requirements Elicitation:** This is also known as gathering of requirements. Here, requirements are identified with the help of customer and existing systems processes, if available.

(*ii*) **Requirements Analysis:** Analysis of requirements starts with requirement elicitation. The requirements are analysed in order to identify inconsistencies, defects, omissions etc. We describe requirements in terms of relationships and also resolve conflicts, if any.

(*iii*) **Requirements Documentation:** This is the end product of requirements elicitation and analysis. The documentation is very important as it will be the foundation for the design of the software. The document is known as software requirements specification (SRS).

(*iv*) **Requirements Review:** The review process is carried out to improve the quality of the SRS. It may also be called as requirements verification. For maximum benefits, review and verification should not be treated as a discrete activity to be done only at the end of the preparation of SRS. It should be treated as continuous activity that is incorporated into the elicitation, analysis, and documentation.

The primary output of requirements engineering is requirements specifications. If it describes both hardware and software, it is a system requirements specification. If it describes only software, it is a software requirements specification. In either case, a requirements specification must treat the system as a black box. It must delineate inputs, outputs, the functional requirements that show external behaviour in terms of input, output, and their relationships, and nonfunctional requirements and their constraints, including performance, portability, and reliability.

The software requirements specification (SRS) should be internally consistent; consistent with existing documents; correct and complete with respect to satisfying needs; understandable to users, customers, designers, and testers; and capable of serving as a basis for both design and test. This SRS document may act as contract between the developer and customer. If developer fails to implement full set of requirements, it may amount to failure in implementing the contracted system.

3.1.2 Present State of Practice

Most software development organizations agree to the fact that there should be a set of activities called requirements engineering and their success is vital to the success of the entire project. So why is the state of the practice no better than it is? There are several reasons, not all of them obvious [BERR98, DAV194, HSIA93]; and some are discussed below:

1. Requirements are difficult to uncover: Today we are automating virtually every kind of task-some that were previously done manually and some that have never been done before. In either kind of application. it is difficult, if not impossible to identify all the requirements, regardless of the techniques we use. No one can see a brand new system in its entirety. Even if someone could, the description is always incomplete at start. Users and developers must resort to trial and error to identify problems and solutions.

2. Requirements change: Because no user can come up with a complete list of requirements at the outset, the requirements get added and changed as the user begins to understand the system and his or her real needs. That is why we always have requirement changes. But, project schedule is seldom adjusted to reflect these modifications. Fluid requirements make it difficult to establish a baseline from which to design and test. Finally, it is hard to justify spending resources to make a requirement specification "perfect", because it will soon change anyway. This is the biggest problem, and there is as yet no technology to overcome it. This problem is often used as an excuse to either eliminate or scale back requirements engineering effort.

3. Over-reliance on CASE tools: Computer Aided Software Engineering (CASE) tools are often sold as panaceas. These exaggerated claims, have created a false sense of trust, which could inflict untold damage on the Software Industry. CASE tools are as important to developers (including requirement writers) as word processors are to authors. However, we must not rely on requirements engineering tools without first understanding and establishing requirements engineering principles, techniques and processes. Furthermore, we must have realistic expectations from the tools.

4. Tight project schedule: Because of either lack of planning or unreasonable customer demand, many projects start with insufficient time to do a decent job. Sometimes, even the allocated time is reduced while the project is under way. It is also customary to reduce time set apart to analyze requirements, for early start of designing and coding, which frequently leads to disaster.

5. Communication barriers: Requirement engineering is communication intensive activity. Users and developers have different vocabularies, professional backgrounds, and tastes. Developers usually want more precise specifications while users prefer natural language. Selecting either results in misunderstanding and confusion.

6. Market-driven software development: Many of the software development is today market driven, developed to satisfy anonymous customers and to keep them coming back to buy upgrades.

7. Lack of resources: There may not be enough resources to build software that can do everything the customer wants. It is essential to rank requirements so that, in the face of pressure to release the software quickly, the most important can be implemented first.

Requirement problems are expensive and plague almost all systems and software development organizations. In most cases, the best we can hope for it is to detect errors in the requirements in time to contain them before the software is released [SAWY99]. Because of the concern for public safety, reputation and capital investments; developers began to recognize the need for clear, concise and complete requirements [COUN99].

3.1.3 Type of Requirements

There are different types of requirements such as:

(*i*) Known requirements—Something a stakeholder believes to be implemented.

(*ii*) Unknown requirements—Forgotten by the stakeholder because they are not needed right now or needed only by another stakeholder.

(*iii*) Undreamt requirements—Stakeholder may not be able to think of new requirements due to limited domain knowledge.

The term stakeholder is used to refer to any one who may have some direct or indirect influence on the system requirements. Stakeholder includes end-users who will interact with the system and every one else in an organisation who will be affected by it [SOMM01].

A known, unknown, or undreamt requirement may be functional or nonfunctional. Functional requirements describe what the software has to do. They are often called product features.

Non-functional requirements are mostly quality requirements that stipulate how well the software does what it has to do. Non functional quality requirements that are especially important to users include specifications of desired performance, availability, reliability, usability and flexibility. Non functional requirements for developers are maintainability, portability, and testability.

Some requirements are architectural, such as component-naming compatibility, interfaceability, upgradability, etc. Other requirements are constraints, such as system design constraints, standards conformance, legal issues and organisational issues. Constraints can come from users or organisations and may be functional or non-functional [ROBE02].

Example 3.1: *A university wishes to develop a software system for the student result management of its M. Tech. Programme. A problem statement is to be prepared for the software development company. The problem statement may give an overview of the existing system and broad expectation from the new software system.*

Solution: The problem statement is prepared by the Examination division of the University and in given below:

"A University conducts a 4-semester M. Tech programme. The students are offered four theory papers and two Lab papers (practicals) during Ist, IInd and IIIrd semesters. The theory

papers offered in these semesters are categorized as either 'Core' or 'Elective'. Core papers do not have an alternative subject, whereas elective papers may have two or more alternative subjects. Thus a student can study any subject out of the choices available for an elective paper.

In Ist, IInd and IIIrd semesters, 2 core papers and 2 elective papers are offered to each student. The students are also required to submit a term paper minor project in IInd and IIIrd semesters each. In IVth semester the students have to give a seminar and submit a dissertation on a topic/subject area of their interest.

The evaluation of each subject is done out of 100 marks. During the semester, minor exams are conducted for each semester. Students are also required to submit assignments as directed by the corresponding faculty and maintain Lab records for practicals. Based on the students' performance in minor exams, assignments, Lab records and their attendance, marks out of 40 are given in each theory paper and practical paper. These marks out of 40 account for internal evaluation of the students. At the end of each semester, major exams are conducted in each subject (theory as well as practical). These exams are evaluated out of 60 marks and account for external evaluation of the students. Thus, the total marks of a student in a subject are obtained by adding the marks obtained in internal and external evaluation.

Every subject has some credit points assigned to it. If the total marks of a student are >= 50 in a subject, he/she is considered 'Pass' in that subject otherwise the student is considered 'Fail' in that subject. If a student passes in a subject he/she earns all the credit points assigned to that subject, but if the students fails in a subject he/she does not earn any credit point in that subject. At any time, the latest information about subjects being offered in various semesters and their credit points can be obtained from University Website.

It is required to develop a system that will manage information about subjects offered in various semesters, students enrolled in various semesters, elective (s) opted by various students in different semesters, marks and credit points obtained by students in different semesters. The system should also have the ability to generate printable mark sheets for each student. Semester-wise detailed mark lists and student performance reports also need to be generated.

Example 3.2: *A university wishes to develop a software system for library management activities. Design the problem statement for the software company.*

Solution: The problem statement prepared by the library staff of the university is given below. Here activities are explained point wise rather than paragraph wise.

A Software has to be developed for automating the manual library system of a University. The system should be standalone in nature. It should be designed to provide functionalities as explained below:

1. Issue of Books:

(*a*) A student of any course should be able to get books issued.

(*b*) Books from **General section** are issued to all but **Book bank** books are issued only for their respective courses.

(*c*) A limitation is imposed on the number of books a student can be issued.

(*d*) A maximum of 4 books from Book bank and 3 books from General section per student is allowed.

(*e*) The books from Book bank are issued for entire semester while books from General section are issued for 15 days only.

(*f*) The software takes the current system date as the date of issue and calculates the corresponding date of return.

(*g*) A bar code detector is used to save the student information as well as book information.

(*h*) The due date for return of the book is stamped on the book.

2. Return of Books:

(*a*) Any person can return the issued books.

(*b*) The student information is displayed using the bar code detector.

(*c*) 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.

(*d*) The system operator verifies the duration for the issue and if the book is being returned after the specified due date, a fine of Re 1 is charged for each day.

(*e*) The information is saved and the corresponding updations take place in the database.

3. Query Processing:

(*a*) The system should be able to provide information like:

(*i*) Availability of a particular book

(*ii*) Availability of books of any particular author.

(*iii*) Number of copies available of the desired book.

(*b*) The system should be able to reserve a book for a particular student for 24 hrs if that book is not currently available.

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.

3.2 REQUIREMENTS ELICITATION

Requirements elicitation is perhaps the most difficult, most critical, most error-prone, and most communication intensive aspect of software development. Elicitation can succeed only through an effective customer-developer partnership [WIEG99].

The real requirements actually reside in user's mind. Hence the most important goal of requirement engineering is to find out what users really need. Users need can be identified only if we understand the expectations of the users from the desired software.

It is the activity that helps to understand the problem to be solved. Requirements are gathered by asking questions, writing down the answers, asking other questions, etc. Hence, requirements gathering is the most communications intensive activity of software development. Developers and Users have different mind set, expertise and vocabularies. Due to communication gap, there are chances of conflicts that may lead to inconsistencies, misunderstanding and omission of requirements.

Therefore, requirements elicitation requires the collaboration of several groups of participants who have different background. On the one hand, customers and users have a solid background in their domain and have a general idea of what the software should do. However, they may have little knowledge of solfware development processes. On the other hand, the developers have experience in developing software but may have little knowledge of everyday environment of the users. Moreover each group may be using incompatible terminologies.

There are number of requirements elicitation methods and few of them are discussed in the following sections. Some people think that one methodology is applicable to all situations, however, generally speaking, one methodology cannot possibly be sufficient for all conditions [MACA96]. We select a particular methodology for the following reason(s):

(*i*) It is the only method that we know.

(*ii*) It is our favorite method for all situations.

(*iii*) We understand intuitively that the method is effective in the present circumstances.

Clearly, third reason demonstrates the most maturity and leads to improved understanding of stakeholder's needs and thus resulting system will satisfy those needs. Unfortunately, most of us do not have the insight necessary to make such an informed decision, and therefore rely on the first two reasons [HICK03].

3.2.1 Interviews

After receiving the problem statement from the customer, the first step is to arrange a meeting with the customer. During the meeting or interview, both the parties would like to understand each other. Normally specialised developers, often called 'requirement engineers' interact with the customer. The objective of conducting an interview is to understand the customer's expectations from the software. Both parties have different feelings, goals, opinions, vocabularies, understandings, but one thing is common, both want the project to be a success. With this in mind, requirement engineers normally arrange interviews. Requirement engineers must be open minded and should not approach the interview with pre-conceived notions about what is required.

Interview may be open-ended or structured. In open-ended interview, there is no pre-set agenda. Context free questions may be asked to understand the problem and to have an overview of the situation. For example, for a "result management system", requirement engineer may ask:

- Who is the controller of examination ?
- Who has requested for such a software ?
- How many officers are placed in the examination division ?
- Who will use the software ?
- Who will explain the manual system ?

- Is there any opposition for this project ?
- How many stakeholders are computer friendly ?

Such questions help to identify all stakeholders who will have interest in the software to be developed.

In structured interview, agenda of fairly open questions is prepared. Sometimes a proper questionnaire is designed for the interview. Interview may be started with simple questions to set people at ease. After making atmosphere comfortable and calm, specific questions may be asked to understand the requirements. The customer may be allowed to voice his or her perceptions about a possible solution.

Selection of Stakeholder. It will be impossible to interview every stakeholder. Thus, representatives from groups must be selected based on their technical expertise, domain knowledge, credibility, and accessibility. There are several groups to be considered for conducting interviews:

(*i*) Entry level personnel: They may not have sufficient domain knowledge and experience, but may be very useful for fresh ideas and different views.

(*ii*) Mid-level stakeholders: They have better domain knowledge and experience of the project. They know the sensitive, complex and critical areas of the project. Hence, requirement engineers may be able to extract meaningful and useful information. Project leader should always be interviewed.

(*iii*) Managers or other Stakeholders: Higher level management officers like vice-Presidents, General Managers, Managing Directors should also be interviewed. Their expectations may provide different but rich information for the software development.

(*iv*) Users of the software: This group is perhaps the most important because they will spend more time interacting with the software than any one else. Their information may be eye opener and may be original at times. Only caution required is that they may be biased towards existing systems.

Types of questions: Questions should be simple and short. Two or three questions rolled into one can lead to compound requirements statements that are difficult to interpret and test. It is important to prepare questions, but reading from the questionnaire or only sticking to it is not desirable. We should be open for any type of discussion and any direction of the interview. For the "result management system" we may ask:

- Are there any problems with the existing system ?
- Have you faced calculation errors in past ?
- What are the possible reasons of malfunctioning ?
- How many students are enrolled presently ?
- What are the possible benefits of computerising this system ?
- Are you satisfied with current processes and policies ?
- How are you maintaining the records of previous students ?
- What data, required by you, exists in other systems ?
- What problems do you want this system to solve ?

- Do you need additional functionality for improving the performance of the system ?
- What should be the most important goal of the proposed development ?

These questions will help to start the communication that is essential for understanding the requirements. At the end of this, we may have wide variety of expectations from the proposed software.

3.2.2 Brainstorming Sessions

Brainstorming is a group technique that may be used during requirements elicitation to understand the requirements. The group discussions may lead to new ideas quickly and help to promote creative thinking.

It is intended to generate lots of ideas, with full understanding that they may not be useful. The theory is that having a long list of requirements from which to choose is far superior to starting with a blank slate. Requirements in the long list can be categorized, prioritized, and pruned [ROBE02].

Brainstorming has become very popular and is being used by most of the companies. It promotes creative thinking, generates new ideas and provides plateform to share views, apprehensions expectations and difficulties of implementation. All participants are encouraged to say whatever ideas come to mind, whether they seem relevant or not. No one will be criticized for any idea, no matter how goofy it seems, as the responsibility of the participant is to generate views and not to vet them.

This group technique may be carried out with specialised groups like actual users, middle level managers etc., or with total stakeholders. Sometimes unnatural groups are created that may not be appriciated and are uncomfortable for participants. At times, only superficial responses may be gathered to technical questions. In order to handle such situations, a highly trained facilitator may be required. The facilitator may handle group bias and group conflicts carefully. The facilitator should also be cautious about individual egos, dominance and will be responsible for smooth conduct of brainstorming sessions. He or she will encourage the participants, ensure proper individual and group behaviour and help to capture the ideas. The facilitator will follow a published agenda and restart the creative process if it falters.

Every idea will be documented in such a way that everyone can see it. White boards, overhead transparencies or a computer projection system can be used to make it visible to every participant. After the session, a detailed report will be prepared and facilitator will review the report. Every idea will be written in simple english so that it conveys same meaning to every stakeholder. Incomplete ideas may be listed separately and should be discussed at length to make them complete ideas, if possible. Finally, a document will be prepared which will have list of requirements and their priority, if possible.

3.2.3 Facilitated Application Specification Technique

This approach is similar to brainstorming sessions and the objective is to bridge the expectation gap – a difference between what developers think they are supposed to build and what customers think they are going to get. In order to reduce expectation gap, a team oriented approach is developed for requirements gathering and is called Facilitated Application Specification Technique (FAST).

This approach encourages the creation of a joint team of customers and developers who work together to understand the expectations and propose a set of requirements. The basic guidelines for FAST are given below:

- Arrange a meeting at a neutral site for developers and customers.
- Establishment of rules for preparation and participation.
- Prepare an informal agenda that encourages free flow of ideas.
- Appoint a facilitator to control the meeting. A facilitator may be a developer, a customer, or an outside expert.
- Prepare a definition mechanism-Board, flip charts, worksheets, wall stickies, etc.
- Participants should not criticize or debate.

FAST session preparations

Each FAST attendee is asked to make a list of objects that are:

(*i*) part of the environment that surrounds the system

(*ii*) produced by the system

(*iii*) used by the system.

In addition, each attendee is asked to make another list of services (processes or functions) that manipulate or interact with the objects. Finally, lists of constraints (*e.g.*, cost, size) and performance criteria (*e.g.*, speed, accuracy) are also developed. The attendees are informed that the lists are not expected to be exhaustive but are expected to reflect each person's perception of the system [PRES2K].

Activities of FAST session

The activities during FAST session may have the following steps:

- Each participant presents his or her lists of objects, services, constraints, and performance for discussion. Lists may be displayed in the meeting by using board, large sheet of paper or any other mechanism, so that they are visible to all the participants.
- The combined lists for each topic are prepared by eliminating redundant enteries and adding new ideas.
- The combined lists are again discussed and consensus lists are finalised by the facilitator.
- Once the consensus lists have been completed, the team is divided into smaller subteams, each works to develop mini-specifications for one or more enteries of the lists.
- Each subteam then presents mini-specifications to all FAST attendees. After discussion, additions or deletions are made to the lists. We may get new objects, services, constraints, or performance requirements to be added to original lists.
- During all discussions, the team may raise an issue that cannot be resolved during the meeting. An issues list is prepared so that these ideas will be considered later.
- Each attendee prepares a list of validation criteria for the product/system and presents the list to the team. A consensus list of validation criteria is then created.

- A subteam may be asked to write the complete draft specifications using all inputs from the FAST meeting.

FAST is not a panacea of the problems encountered in early requirements elicitation but it helps to understand the requirements and bridge the expectation gap of develops and customers.

3.2.4 Quality Function Deployment

It is a quality management technique that helps to incorporate the voice of the customer. The voice is then translated into technical requirements. These technical requirements are documented and results is the software requirements and specification document. These requirements are further translated into design requirements. Here, customer satisfaction is of prime concern and thus QFD emphasizes an understanding of what is valuable to the customer and then deploys these values throughout the software engineering process [PRES2K]. Three types of requirements are identified [ZULT92]:

(*i*) **Normal requirements.** The objectives and goals of the proposed software are discussed with the customer. If this category of requirements (normal) are present, the customer is satisfied. Examples related to result management system might be: entry of marks, calculation of results, merit list report, failed students report, etc.

(*ii*) **Expected requirements.** These requirements are implicit to the software product and may be so obvious that customer does not explicitly state them. If such requirements are not present, customer will be dissatisfied with the software. Examples of expected requirements may be: protection from unauthorised access, some warning system for wrong entry of data, the feasibility for modification of any record only by a fool proof system for the identification of person alongwith date and time of modification, etc.

(*iii*) **Exciting requirements.** Some features go beyond the customer's expectations and prove to be very satisfying when present. Examples of exciting requirements for result management system may be: if an unauthorised access is noticed by the software, it should immediately shutdown all the processes and an E-mail is generated to the system administrator, an additional copy of important files is maintained and may be accessed by system administrator only, sophisticated virus protection system etc.

The QFD method has the following steps [ROBE02]:

(*i*) Identify all the stakeholders *e.g.*, customers, users, and developers. Also identify any initial constraints identified by the customer that affect requirements development.

(*ii*) List out requirements from customer ; inputs, considering different viewpoints. Requirements are expression of what the system will do, which is both perceptible and of value to customers. Some customer's expectations may be unrealistic or ambiguous and may be translated into realistic or unambiguous requirements if possible.

(*iii*) A value indicating a degree of importance, is assigned to each requirement. Thus, customer determines the importance of each requirement on a scale of 1 to 5 as given below:

5 points:	Very important
4 points:	important

3 points: not important, but nice to have
2 points: not important
1 point: unrealistic, requires further exploration

Stakeholders will have their own unique set of criteria for determining the 'importance', or 'value' *of a requirement. It may be based on cost/benefit analysis particular to the project.*

Requirement Engineers may review the final list of requirements and categorise like:

(*i*) it is possible to achieve

(*ii*) it should be deferred and the reason thereof

(*iii*) it is impossible and should be dropped from consideration.

The first category requirements will be implemented as per priority (Importance value) assigned with every requirement. If time and effort permits, second category requirements may be reviewed and few of them may be transferred to category first for implementation.

3.2.5 The Use Case Approach

For many years, requirement engineers have used stories or scenarios to explain the interaction of a user with the proposed software system in order to gather the requirements. More recently, Ivar Jacobson and others [JACO99] formalised this into use case approach to requirements elicitation and modeling. Initially, use cases were designed for object oriented software development world, however, they can be applied to any project that follow any development approach because the user does not care how we develop the software. The focus on what the users need to do with the system is much more powerful than the traditional elicitation approach of asking users what they want the system to do [WIEG99].

This approach uses a combination of text and pictures in order to improve the understanding of requirements. The Use cases describe what of a system and not 'how'. They only give functional view of the system.

The terms use case, use case scenario, and use case diagram are often interchanged, but in fact they are different. Use cases are structured outline or templates for the description of user requirements, modeled in a structured language like english. Use case scenarios are unstructured descriptions of user requirements. Use case diagrams are graphical representations that may be decomposed into further levels of abstraction. The following components are used for the design of the use case approach.

Actor: An actor or external agent, lies outside the system model, but interacts with it in some way. An actor may be a person, machine, or an information system that is external to the system model. An actor is represented as stick figure and is not part of the system itself. Customers, users, external devices, or any external entity interacting with the system are treated as actors.

We should not confuse the actors with the devices they use. Devices are typically mechanisms that actors use to communicate with the system, but they are not actors themselves. We are writing this book on a computer, but the keyboad is not the user of the word processing program; but we are. Other devices, such as disk drives, tape drives, or communication equipment including printers have no place in use case diagram, although they are important to the

design of the system. The purpose of devices is to support some required behaviour of the system, but devices do not define the requirements of the system. Often systems must produce a printed report of information that it contains. We may want to show printer as an actor that then forwards the report to the real actor. This is not correct. Printer is not an actor; it is just a mechanism for conveying information [BITT03].

Cockburn [COCK01] distinguishes between primary and secondary actors. A primary actor is one having a goal requiring the assistance of the system. A secondary actor is one from which the system needs assistance.

Use Cases: A use case is initiated by a user with a particular goal in mind, and completes successfully when that goal is satisfied. It describes the sequence of interactions between actors and the system necessary to deliver the services that satisfies the goal. It also includes possible variants of this sequence, *e.g.,* alternative sequences that may also satisfy the goal, as well as sequences that may lead to failure to complete the service because of exceptional behaviour, error handling etc. The system is treated as a 'black box', and the interactions with the system, including responses, are as perceived from outside the system.

Thus, use cases capture who (actor) does what (interaction) with the system, for what purpose (goal), without dealing with system internals. A complete set of use cases specifies all the different ways to use the system, and therefore, defines all behaviour required of the system, bounding the scope of the system.

Use cases are written in an easy to understand structured narrative—the vocabulary of the domain. The users may validate the use cases and may involve in the process of gathering and defining the requirements [MALA01].

There is no standard use case template for writing use cases. The Jacobson et al. [JACO99] proposed a template for writing use cases and is given in Table 3.1(a). This template captures requirements in an effective way and is therefore becoming popular. Another similar template is also given in Table 3.1(b) which is also used by many organisations.

Use case guidelines

The following provides an outline of a process for creating use cases:

- Identify all the different users of the system.
- Create a user profile for each category of users, including all the roles the users play that are relevant to the system. For each role, identify all the significant goals the users have that the system will support. A statement of the system's value proposition is useful in identifying significant goals.
- Create a use case for each goal, following the use case template. Maintain the same level of abstraction throughout the use case. Steps in higher level use cases may be treated as goals for lower level (*i.e.,* more detailed), sub-use cases.
- Structure the use cases. Avoid over-structuring, as this can make the use cases harder to follow.
- Review and validate with users.

Table 3.1(a): Use case template

1.	**Brief Description.** Describe a quick background of the use case.
2.	**Actors.** List the actors that interact and participate in this use case.
3.	**Flow of Events.** 3.1. ***Basic flow.*** List the primary events that will occur when this use case is executed. 3.2. ***Alternative flows.*** Any subsidiary events that can occur in the use case should be separately listed. List each such event as an alternative flow. A use case can have as many alternative flows as required.
4.	**Special Requirements.** Business rules for the basic and alternative flows should be listed as special requirements in the use case narration. These business rules will also be used for writing test cases. Both success and failure scenarios should be described here.
5.	**Pre-conditions.** Pre-conditions that need to be satisfied for the use case to perform.
6.	**Post-conditions.** Define the different states in which you expect the system to be in, after the use case executes.
7.	**Extension Points.**

Table 3.1(b): Use case template

1.	**Introduction.** Describe brief purpose of the use case.
2.	**Actors.** List the actors that interact and participate in this use case.
3.	**Pre-condition.** Condition that need to be satisfied for the use case to execute.
4.	**Post-condition.** After the execution of the use case, different states of the systems are defined here.
5.	**Flow of Events.** 5.1. ***Basic flow.*** List the primary events that will occur when this use case is executed. 5.2. ***Alternate flow.*** Any other possible flow in this use case, if there, should be separately listed. A use case may have many alternate flows.
6.	**Special Requirements.** Business rules for the basic and alternate flows should be listed as special requirements. Both success and failure scenarios should be described.
7.	**Related use cases.** List the related use cases, if any.

Use case diagrams

A use case diagram visually represents what happens when an actor interacts with a system. Hence, a use case diagram captures the functional aspects of a system. The system is shown as a rectangle with the name of the system (or subsystem) inside, the actors are shown as stick figures (even the non human ones), the use cases are shown as solid bordered ovals labeled with the name of the use case, and relationships are lines or arrows between actors and use cases and/or between the use cases themselves. These components are given in Fig. 3.2.

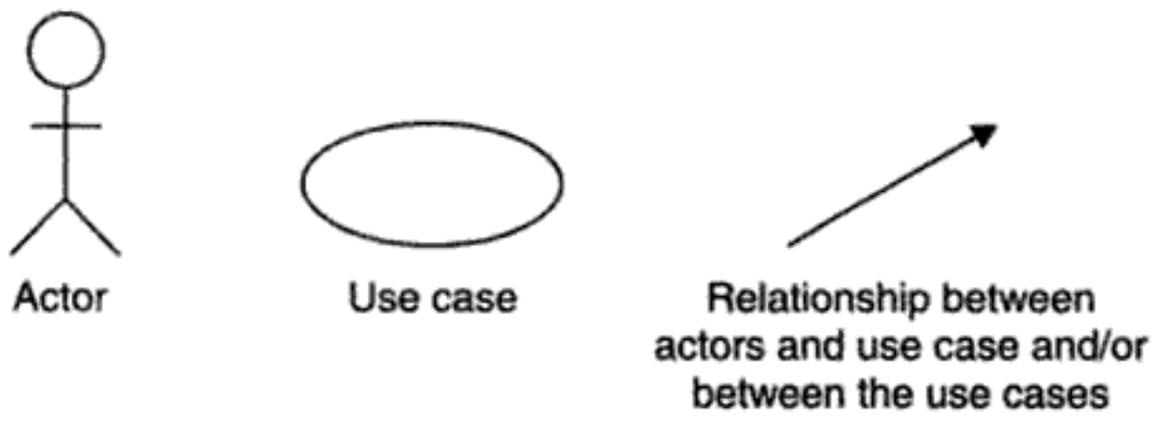


Fig. 3.2: Components of use case diagram.

Actors appear outside of the rectangle since they are external to the system. Use cases appear within the rectangle, providing functionality. A relationship or association is a solid line between an actor and each use case in which actor participates—the involvement can be any kind, not necessarily one of the actor initiating the use case functionality.

Fig. 3.3 shows an example of a use case diagram whose "Problem statement" is given in Table 3.2.

Table 3.2: Problem statement for railway reservation system

PROBLEM STATEMENT FOR RAILWAY RESERVATION SYSTEM

A Software has to be developed for automating the manual railway reservation system. The system should be distributed in nature. It should be designed to provide functionalities as explained below:

1. **Reserve Seat:** A passenger should be able to reserve seats in the train. A reservation form is filled by the passenger and given to the clerk, who then checks for the availability of seats for the specified date of journey. If seats are available, then the entries are made in the system regarding the train name, train number, date of journey, boarding station, destination, person name, sex and total fare. Passenger is asked to pay the required fare and the tickets are printed. It the seats are not available then the passenger is informed.
2. **Cancel Reservation:** A passenger wishing to cancel a reservation is required to fill a form. The passenger then submits the form and the ticket to the clerk. The clerk then deletes the entries in the system and changes in the reservation status of that train. The clerk crosses the ticket by hand to mark as cancelled.
3. **Update Train Information:** Only the administrator enters any changes related to the train information like change in the train name, train number, train route etc. in the system.
4. **Report Generation:** Provision for generation of different reports should be given in the system. The system should be able to generate reservation chart, monthly train report etc.
5. **Login:** For security reasons all the users of the system are given a user *id* and a password. Only if the *id* and password are correct the user is allowed to enter the system.
6. **View Reservation Status:** All the users should be able to see the reservation status of the train online. The user needs to enter the train number and the pin number printed on his ticket so that the system can display his current reservation status like confirmed, RAC or Wait listed.
7. **View Train Schedule:** Prcvision should be given to see information related to the train schedules for the entire train network. The user should be able to see the train name, train number, boarding and destination stations, duration of journey etc.

Use cases should not be used to capture all the details of system. The granularity to which we define use cases in a diagram should be enough to keep the use case diagram uncluttered and readable, yet, be complete without missing significant aspects of the required

functionality. Design issues should not be discussed at all. Use cases are meant to capture "what" the system is, and not "how" the system will be designed or built. Hence use cases should be free of any design characteristics. If we end up defining design characteristics in a use case, we need to go back to the drawing board and start again.

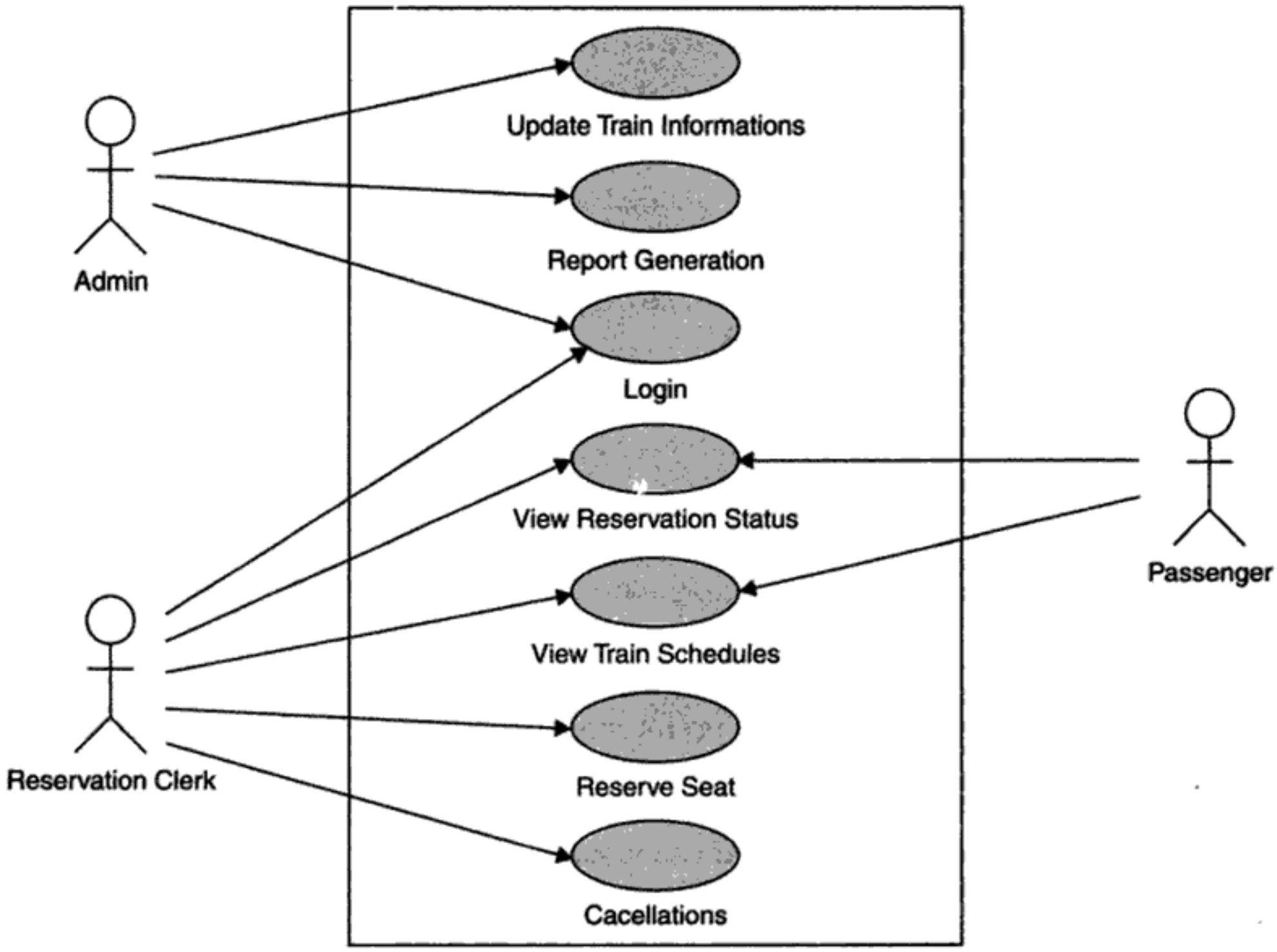


Fig. 3.3: Use Case Diagram for Railway Reservation System.

3.3 REQUIREMENTS ANALYSIS

Requirements analysis is very important and essential activity after elicitation. We analyze, refine and scrutinize the gathered requirements in order to make consistent and unambiguous requirements. This activity reviews all requirements and may provide a graphical view of the

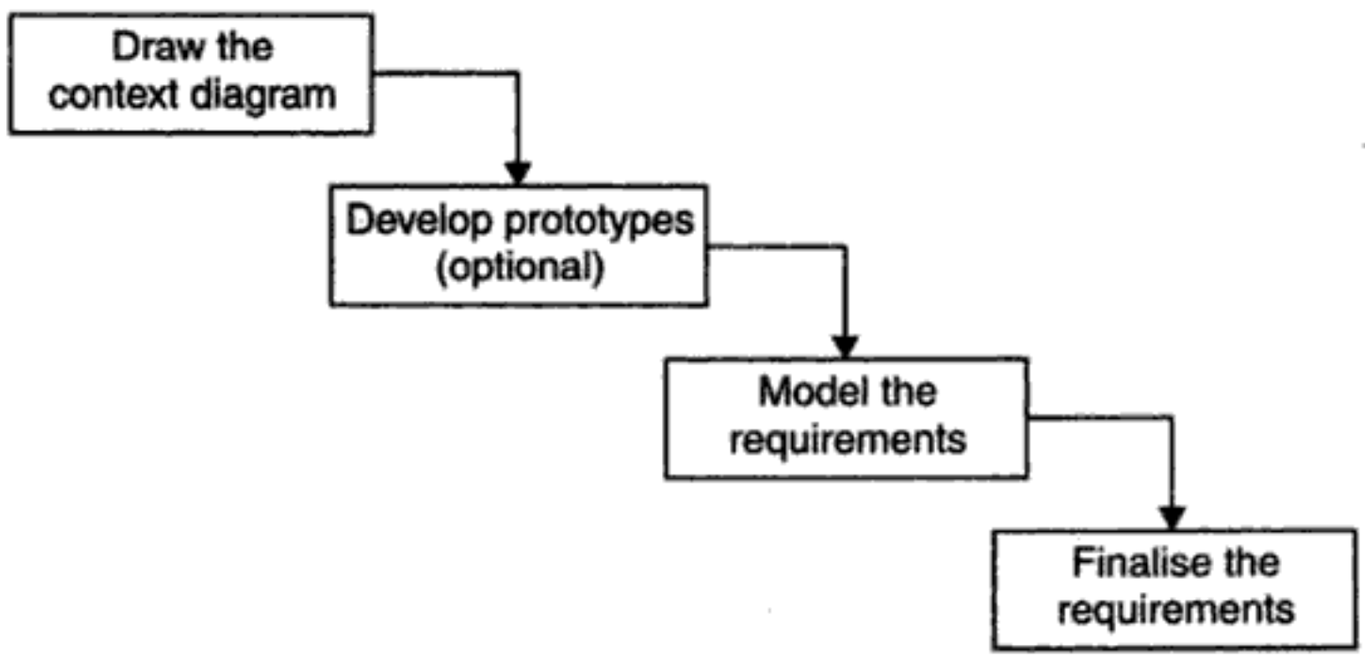
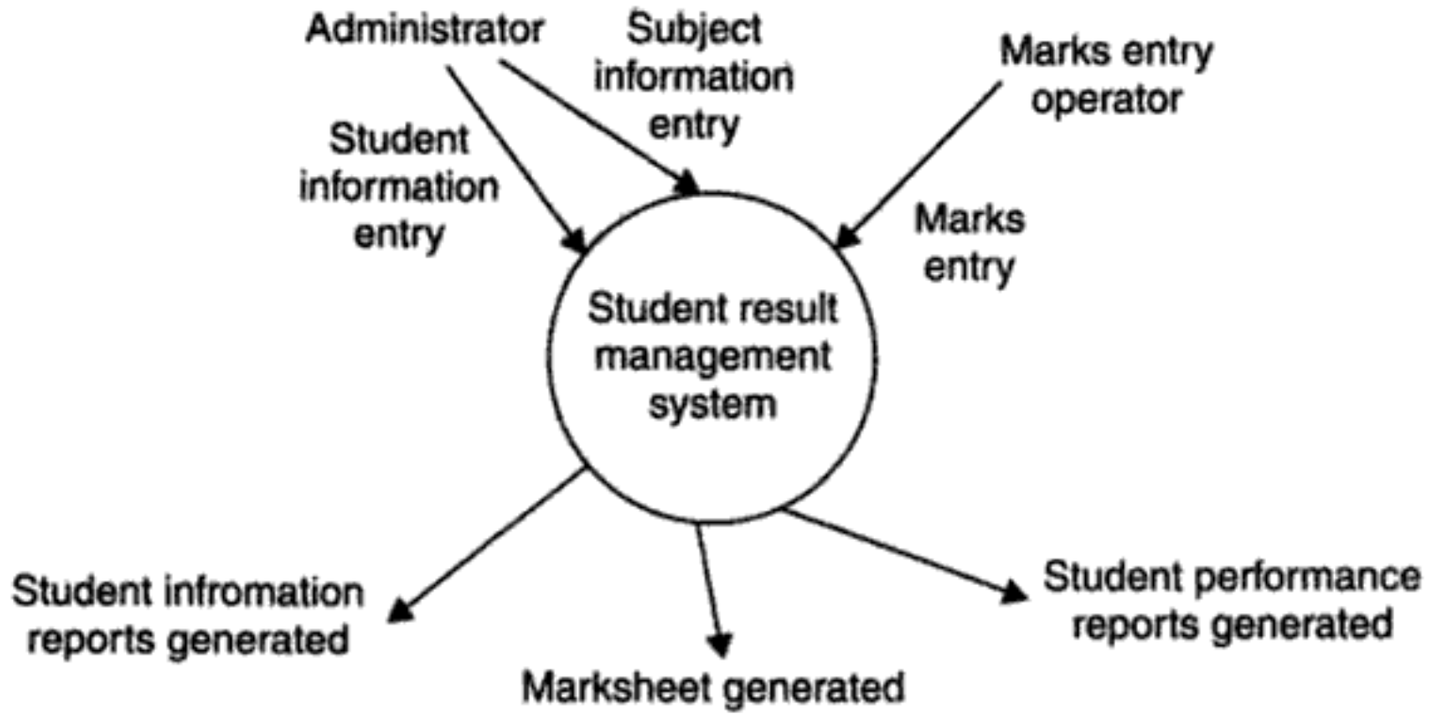


Fig. 3.4: Requirements analysis steps.

entire system. After the completion of analysis, it is expected that the understandability of the project may improve significantly. Here, we may also interact with the customer to clarify points of confusion and to understand which requirements are more important than others. The various steps of requirements analysis are shown in Fig. 3.4.

(*i*) **Draw the context diagram.** The context diagram is a simple model that defines the boundaries and interfaces of the proposed system with the external world. It identifies the entities outside the proposed system that interact with the system. The context diagram of student result management system (as discussed earlier) is given below:



(*ii*) **Development of a prototype (optional).** One effective way to find out what the customer really wants is to construct a prototype, something that looks and preferably acts like a part of the system they say they want.

We can use their feedback to continuously modify the prototype until the customer is satisfied. Hence, prototype helps the client to visualise the proposed system and increase the understanding of requirements. When developers and users are not certain about some of the requirements, a prototype may help both the parties to take a final decision.

Some projects are developed for general market. In such cases, the prototype should be shown to some representative sample of the population of potential purchasers. Even though, persons who try out a prototype may not buy the final system, but their feedback may allow us to make the product more attractive to others. Some projects are developed for a specific customer under contract. On such projects, only that customer's opinion counts, so the prototype should be shown to the prospective users in the customer organisation.

The prototype should be built quickly and at a relatively low cost. Hence it will always have limitations and would not be acceptable in the final system. This is an optional activity. Although many organisations are developing prototypes for better understanding before the finalisation of SRS.

(*iii*) **Model the requirements.** This process usually consists of various graphical representations of the functions, data entities, external entities and the relationships between them. The graphical view may help to find incorrect, inconsistent, missing and superfluous requirements. Such models include data flow diagrams, entity relationship diagrams, data dictionaries, state-transition diagrams etc.

(*iv*) **Finalise the requirements.** After modeling the requirements, we will have better understanding of the system behaviour. The inconsistencies and ambiguties have been identified and corrected. Flow of data amongest various modules has been analysed. Elicitation and analysis activities have provided better insight to the system. Now we finalise the analysed requirements and next step is to document these requirements in a prescribed format.

3.3.1 Data Flow Diagrams

Data flow diagrams (DFD) are used widely for modeling the requirements. They have been used for many years prior to the advent of computers. DFDs show the flow of data through a system. The system may be a company, an organization, a set of procedures, a computer hardware system, a software system, or any combination of the preceding. The DFD is also known as a data flow graph or a bubble chart.

The following observations about DFDs are important [DAV190]:

1. All names should be unique. This makes it easier to refer to items in the DFD.
2. Remember that a DFD is not a flow chart. Arrows in a flow chart represent the order of events; arrows in DFD represent flowing data. A DFD does not imply any order of events.
3. Suppress logical decisions. If we ever have the urge to draw a diamond-shaped box in a DFD, suppress that urge! A diamond-shaped box is used in flow charts to represent decision points with multiple exit paths of which only one is taken. This implies an ordering of events, which makes no sense in a DFD.
4. Do not become bogged down with details. Defer error conditions and error handling until the end of the analysis.

Standard symbols for DFDs are derived from the electric circuit diagram analysis and are shown in Fig. 3.5 [SAGE90].

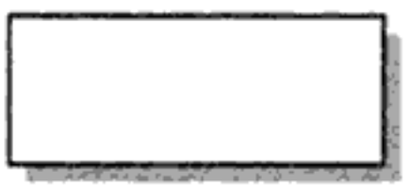
Symbol	*Name*	*Function*
	Data Flow	Used to connect processes to each other, to sources or sinks; the arrowhead indicates direction of data flow.
	Process	Performs some transformation of input data to yield output data.
	Source or Sink (External Entity)	A source of system inputs or sink of system outputs.
	Data Store	A repository of data; the arrowheads indicate net inputs and net outputs to store.

Fig. 3.5: Symbols for data flow diagrams.

A circle (bubble) shows a process that transforms data inputs into data outputs. A curved line shows flow of data into or out of a process or data store. A set of parallel lines shows a place for the collection of data items. A data store indicates that the data is stored which can be used at a later stage or by the other processes in a different order. The data store can have element or group of elements. Source or sink is an external entity and acts as a source of system inputs or sink of system outputs.

Leveling

The DFD may be used to represent a system or software at any level of abstraction. In fact, DFDs may be partitioned into levels that represent increasing information flow and functional detail. A level-0 DFD, also called a fundamental system model or *context diagram* represents the entire software element as a single bubble with input and output data indicated by incoming and outgoing arrows, respectively [PRES2K]. Then the system is decomposed and represented as a DFD with multiple bubbles. Parts of the system represented by each of these bubbles are then decomposed and documented as more and more detailed DFDs. This process may be repeated at as many levels as necessary until the problem at hand is well understood. It is important to preserve the number of inputs and outputs between levels; this concept is called leveling by DeMacro. Thus, if bubble "A" has two inputs, x_1 and x_2, and one output y, then the expanded DFD, that represents "A" should have exactly two external inputs and one external output as shown in Fig. 3.6 [DEMA79, DAV190].

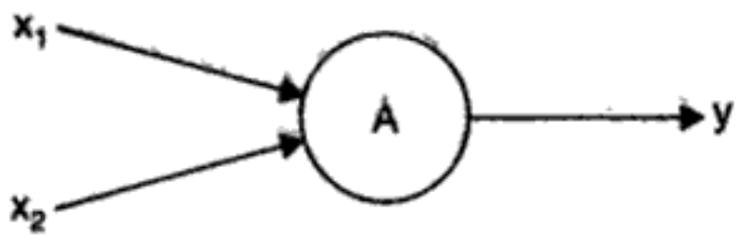
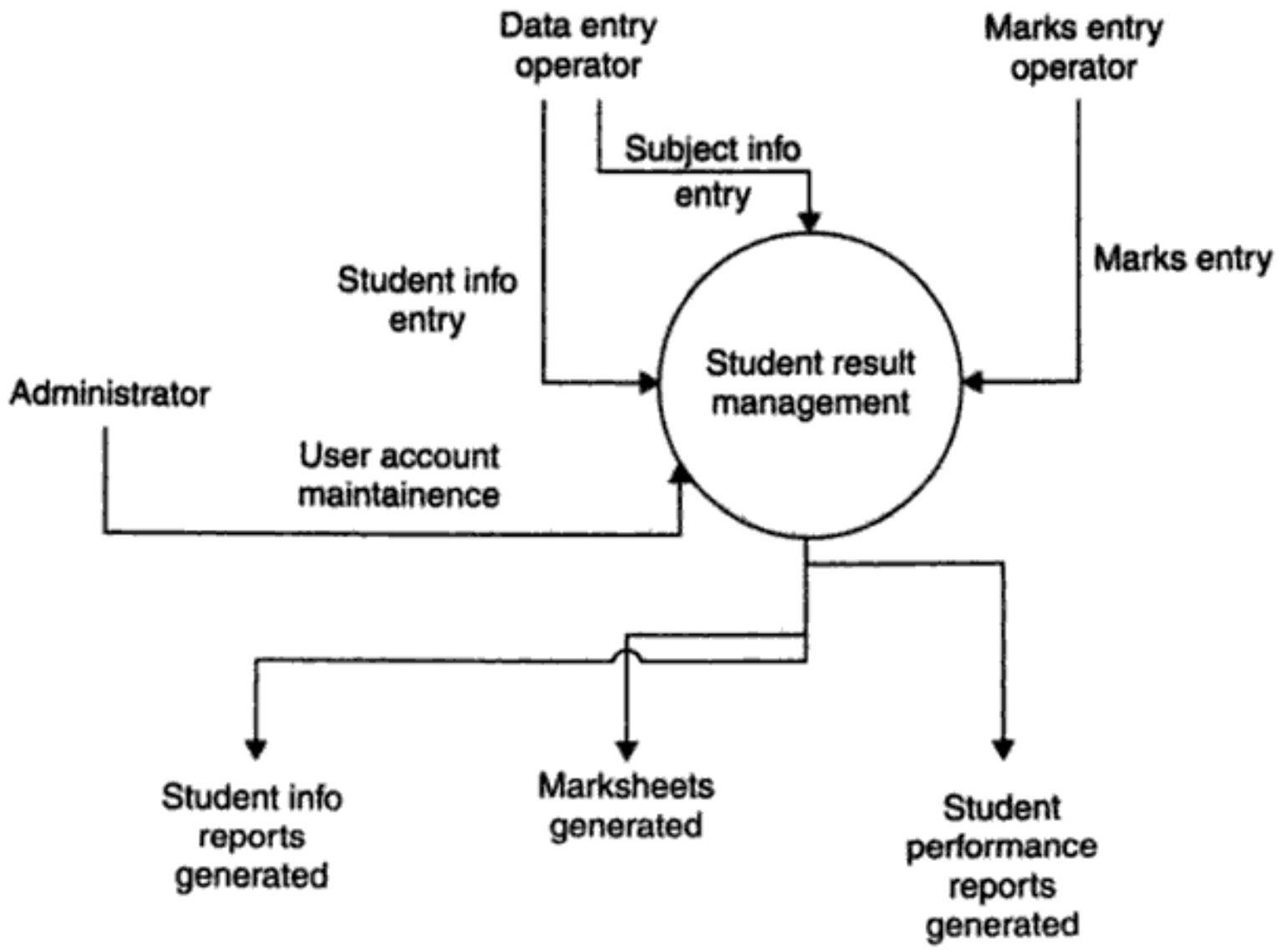


Fig. 3.6: Level-0 DFD.

The level-0 DFD, also called context diagram of result management system is shown in Fig. 3.7. As the bubbles are decomposed into less and less abstract bubbles, the corresponding data flows may also need to be decomposed. Level-1 DFD of result management system is given in Fig. 3.8.

This provides a detailed view of requirements and flow of data from one bubble to the another.



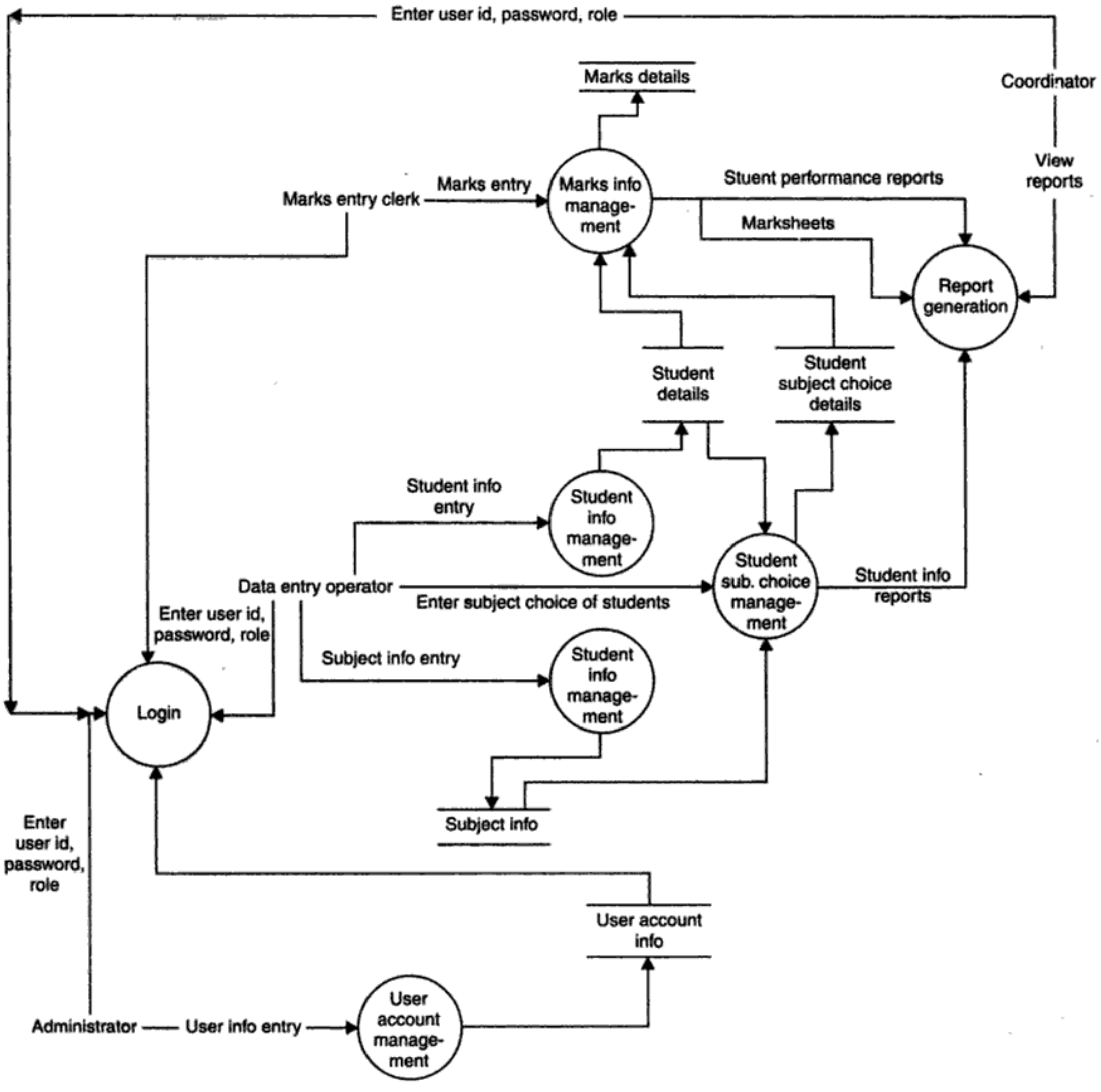


Fig. 3.8: Level-1 DFD of result management system.

3.3.2 Data Dictionaries

Families of DFDs can become quite complex. One way to manage this complexity is to augment DFDs with data dictionaries (DD). Data dictionaries are simply repositories to store information about all data items defined in DFDs. At the requirements stage, the data dictionary should at least define customer data items, to ensure that the customer and developer use the same definitions and terminalogies. Typical information stored includes:

- Name of the data item
- Aliases (other names for item)

- 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.

3.3.3 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 organization and uses three main constructs *i.e.*, data entities, relationships, and their associated attributes.

Entities

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

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

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

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

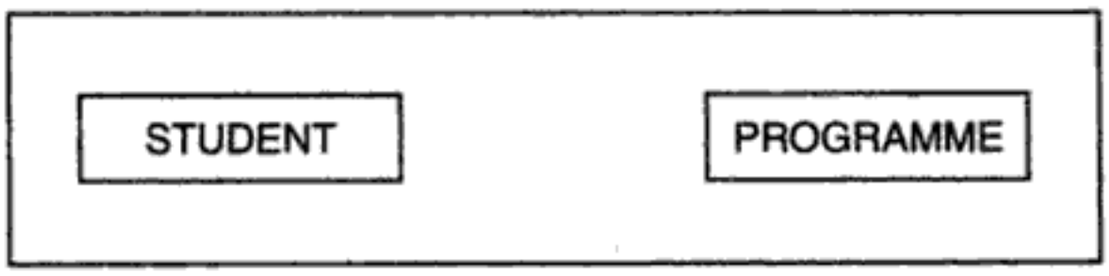


Fig. 3.9: Two entity types in an E-R diagram.

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 PROGRAMME. Relationships are represented by diamond notation in the E-R diagram as shown in Fig. 3.10.

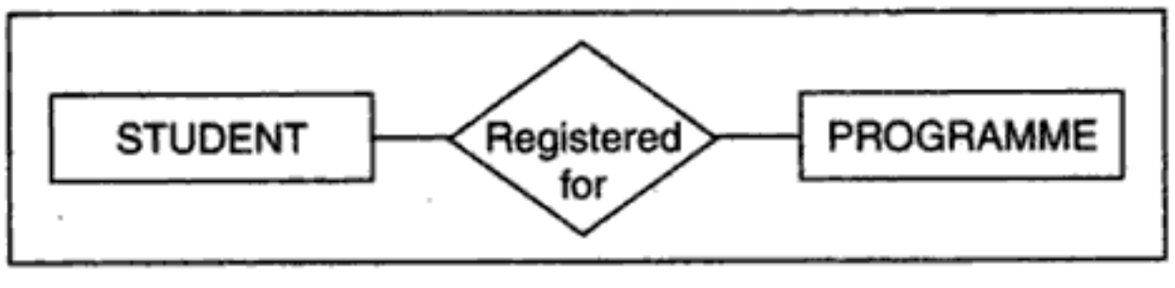


Fig. 3.10: Relationships added to ERD.

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.

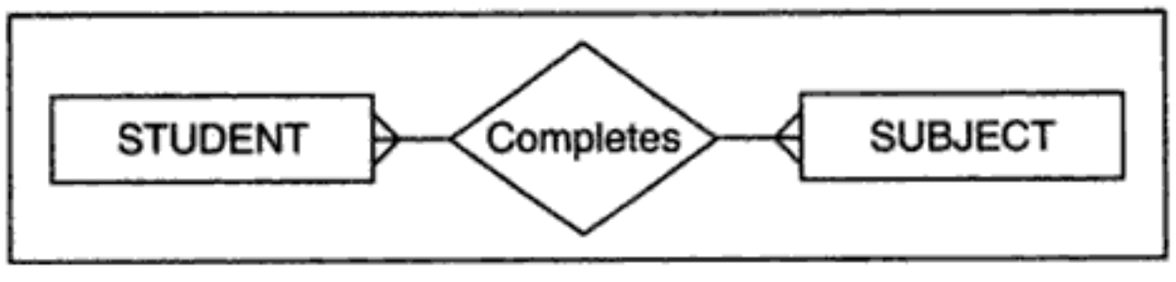


Fig. 3.11: Relationships in ERD.

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.

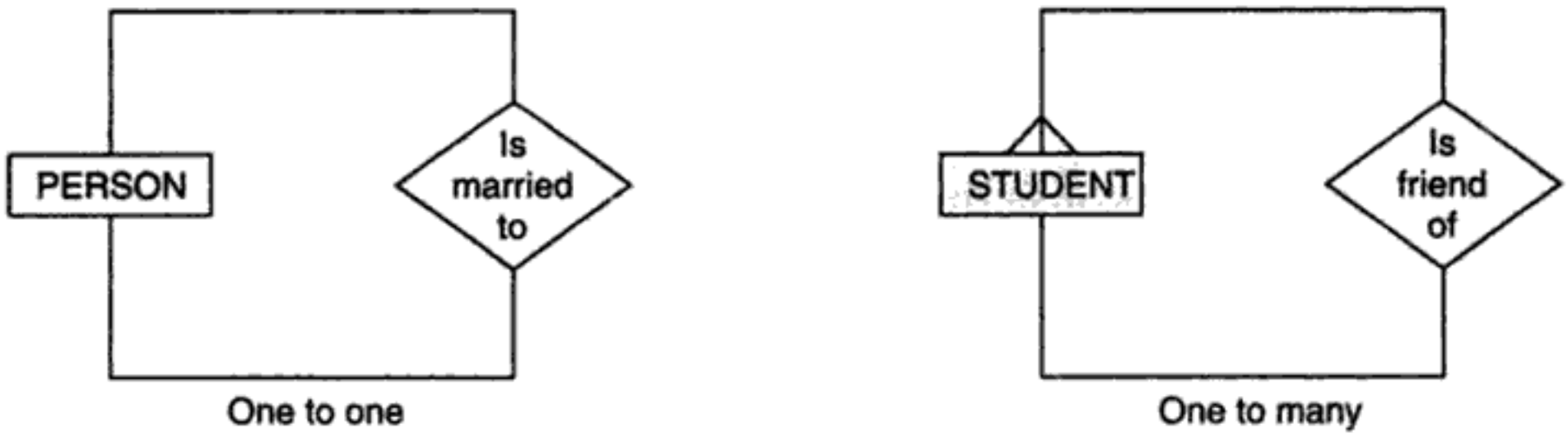


Fig. 3.12: Unary relationships.

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.

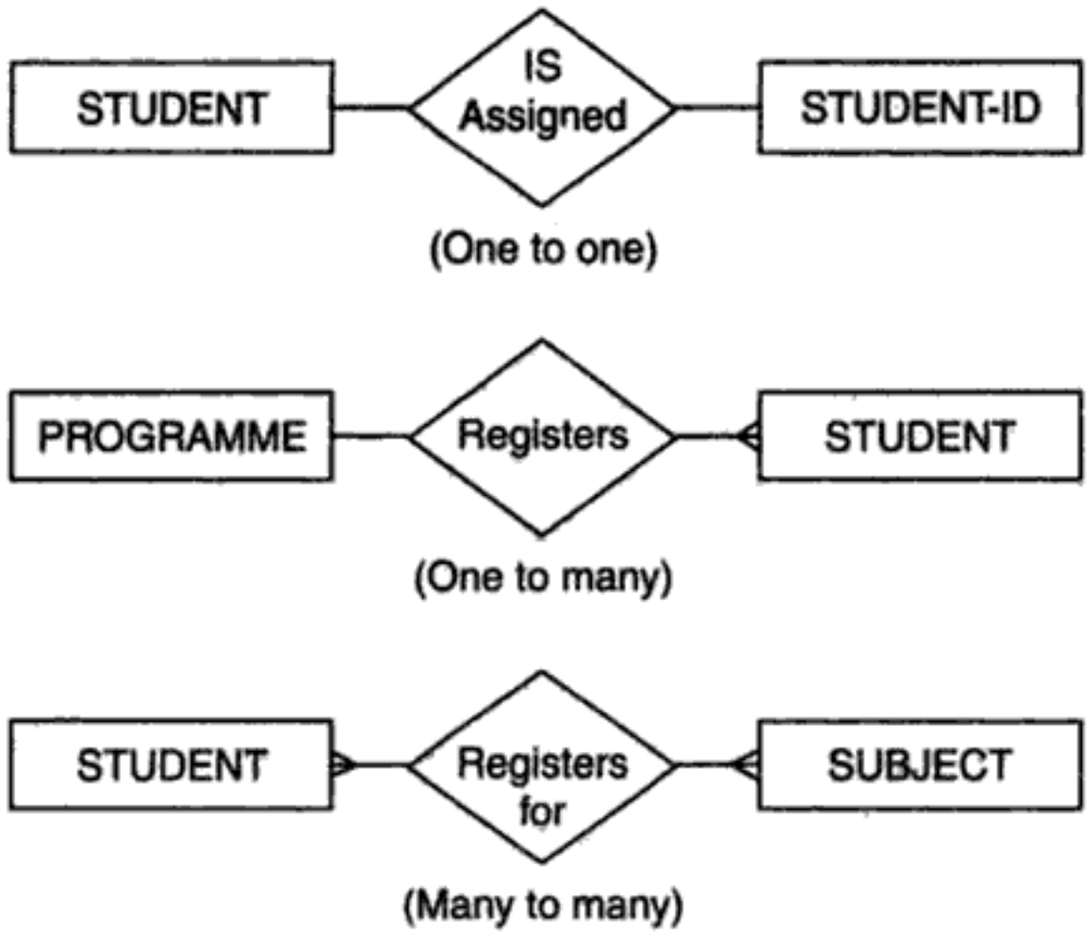


Fig. 3.13: Binary relationships.

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.

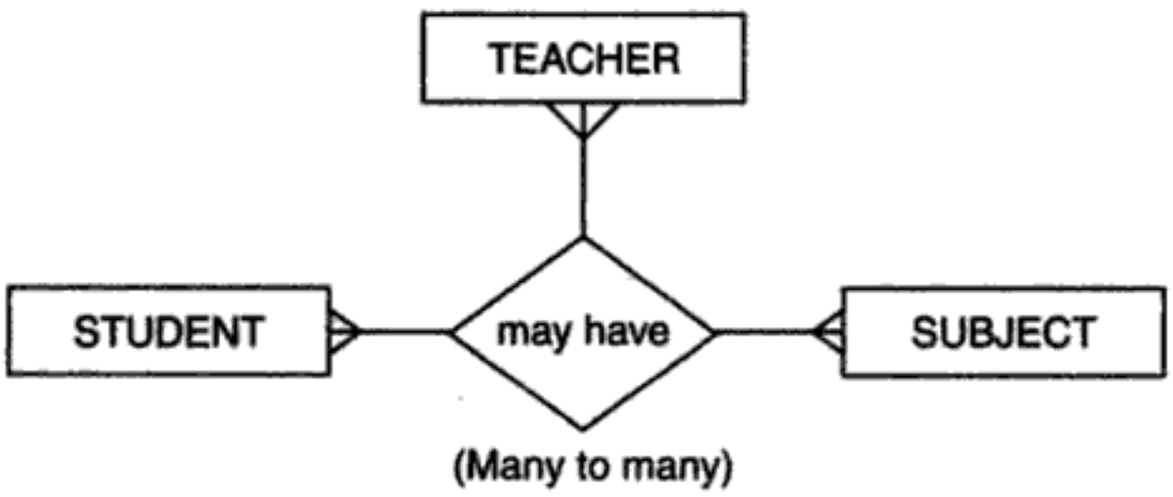


Fig. 3.14: Ternary relationship.

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.

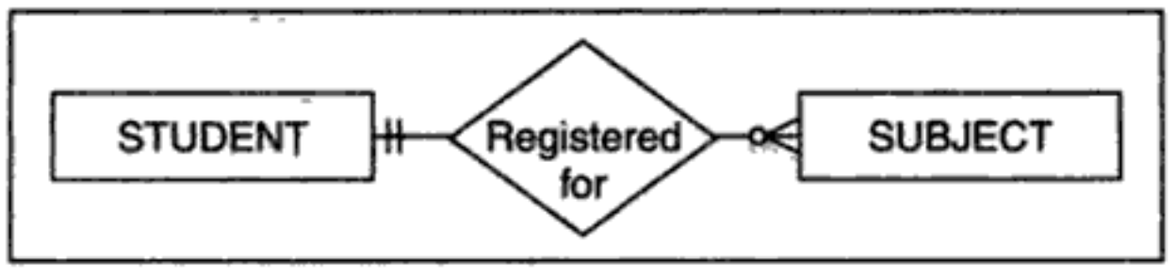


Fig. 3.16: Modified ER diagram.

Cardinality of relationships

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

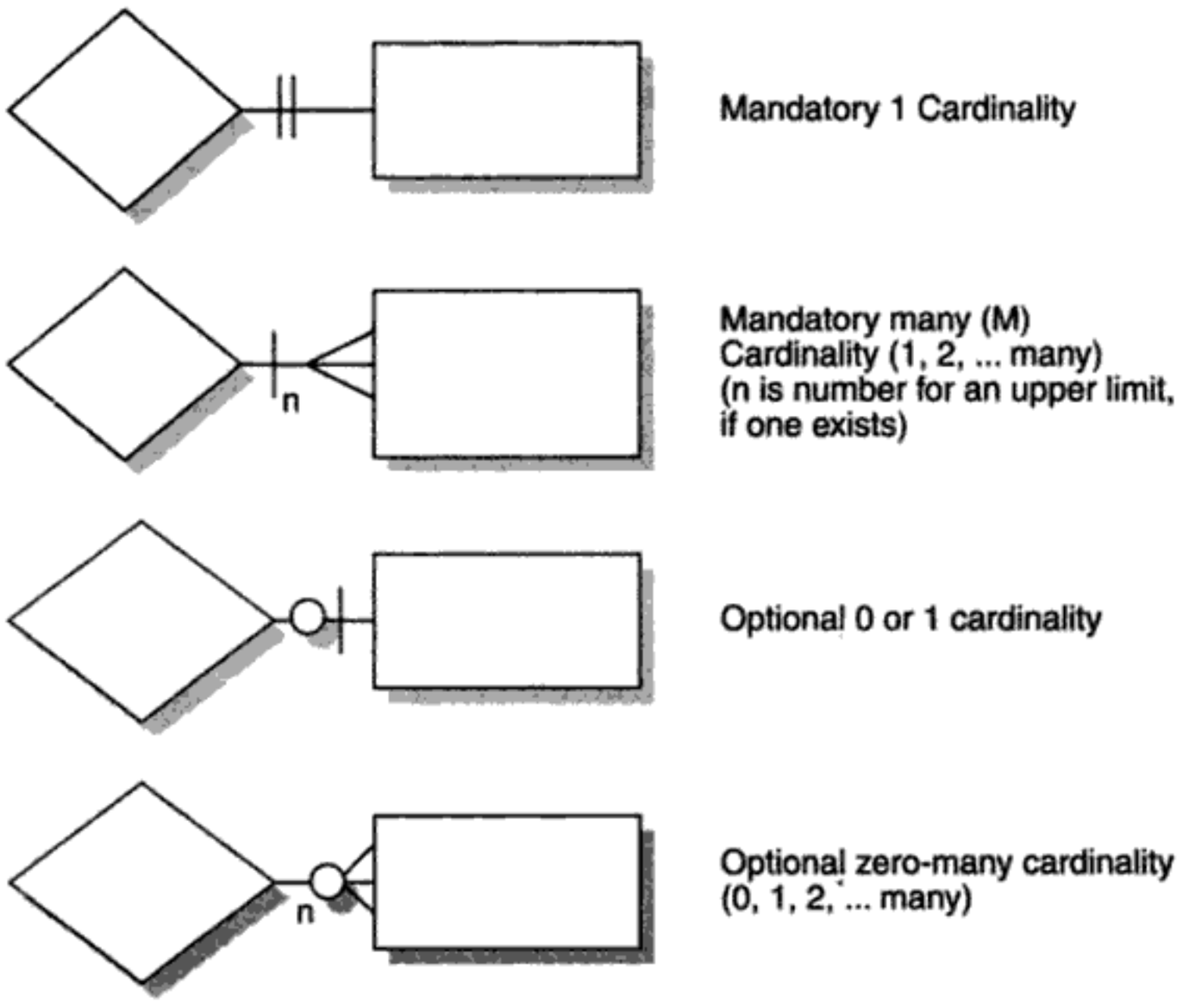


Fig. 3.17: Relationship cardinality [HOFF99]

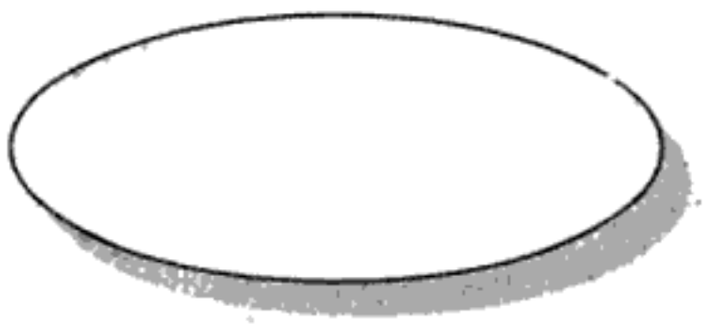
Attributes

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

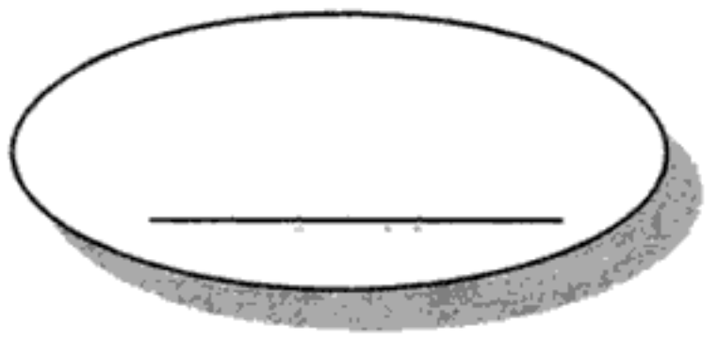


Attribute

Candidate keys and identifier

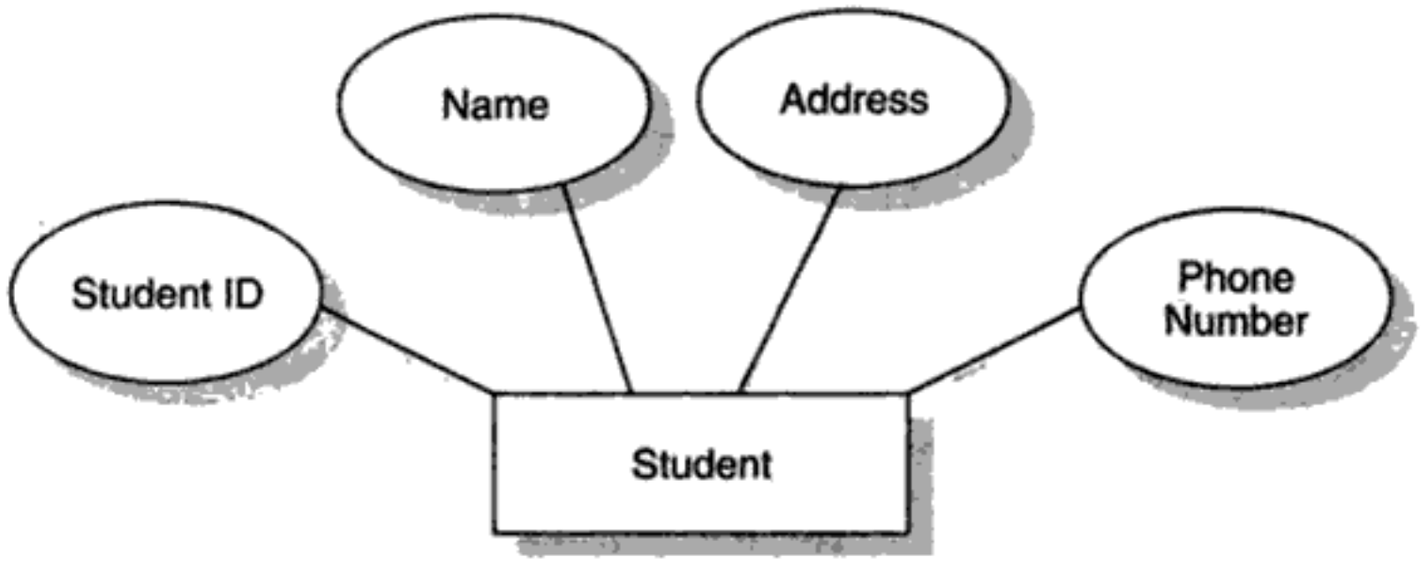
Every entity type must have an attribute or set of attributes that distinguishes one instance from other instances of the same type. A candidate key is an attribute (or combination of attributes) that uniquely identifies each instance of an entity type. A candidate key for a STUDENT entity type might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

3.3.4 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.*, throwaway and evolutionary. In throwaway 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.

4. A working system is available quickly to demonstrate the feasibility and usefulness of the application to management.
5. The prototype serves as a basis for writing the specification of the system.

Software prototyping taxonomy

A range of possibilities exists for prototyping software systems. Any form of prototyping is perceived better than not prototyping at all. Several taxonomies have been proposed and served as basis for prototyping [RATC88, HOOP89, CERI86, CARE90, HEKM87]. However, the two most popular prototyping approaches mentioned earlier are briefly described as:

1. Throw-away prototyping. In this approach, prototype is constructed with the idea that it will be discarded, after the analysis is complete, and the final system is built from the scratch. This prototype is generally built quickly so as to enable the user to rapidly interact with the requirements determination early and thoroughly. Since the prototype will ultimately be discarded, it need not necessarily be fast operating, maintainable and having extensive fault tolerant capabilities.

During the requirement phase, a quick and dirty throwaway prototype can be constructed and given to user in order to determine the feasibility of a requirement, validate that a particular function is really necessary, uncover missing requirements and determine the viability of a user interface. During preliminary and detailed design, a quick and dirty prototype can be built to give a feeling and overview of final system to the user. Here, development of prototype should be quick, because its advantage exists only if results from its use are available in a timely fashion. It can be dirty because there is no justification for building quality into a product that will be discarded. Among the dirty characteristics to be considered are no design, no comments, no test plans, no idea about coupling and cohesion etc.

The most common steps for this approach are: (*i*) Writing a preliminary SRS (*ii*) implementing the prototype based on those requirements (*iii*) achieving user experience with the prototype (*iv*) Writing the real SRS and then (*v*) developing the real product.

2. Evolutionary Prototyping. In this approach, the prototype is built with the idea that it will eventually be converted into the final system. It will not be built in a "dirty" fashion. The evolutionary prototype evolves into the final product, and thus it must exhibit all the quality attributes of the final product and must follow the traditional life cycle. It is required to deploy the product, obtain experience using it, then based on that experience go back and redo the requirements, redesign, recode, retest, and redeploy. After gaining more experience, it is time to repeat the entire process again. This ensures the creation of all necessary documents and the presence of all necessary reviews. In fact, the only shortcuts that should be taken in building evolutionary prototypes are (*i*) building only those parts of the product that are understood (leaving other parts to later generations of the prototype) (*ii*) lowering the importance of performance. Using this will increase the probability that version $i + 1$ will meet user's real needs because users have already used version i and supplied feedback on its performance.

The differences between these two approaches are given below:

Sr. No.	*Approach and Characteristics*	*Throwaway*	*Evolutionary*
1	Development Approach	Quick and Dirty , No rigor	No Sloppiness, Rigorous
2	What to build	Build only difficult parts	Build understood parts first and build on solid foundations
3	Design drivers	Optimize development time	Optimize Modifiability
4	Ultimate Goal	Throw it away	Evolve it

Prototyping pitfalls

Prototyping has not been as successful as anticipated in some organizations for a variety of reasons [TOZE87]. Training, efficiency, applicability, and behaviour can each have a negative impact on using software prototyping techniques.

A common problem with adopting prototyping technology is high expectations for productivity with insufficient effort. Prototyping can have execution inefficiencies with the associated tools and this question may be argued as a negative aspect of prototyping.

This new approach of providing feedback early to the end user may result in a problem related to the behavior of the end user and developers. An end user with a previously unfortunate system development effort can be biased in future interactions with development teams.

Prototyping opportunities

Not to prototype at all should simply not be an option in software development. The benefits of software prototyping are obvious and established. The end user cannot throw the ambiguous and incomplete software needs and expect the development team to return the finished software system after some period of time with no problems in the deliverables.

One of the major problems incorporating this technology is the large investment that exists in software system maintenance. The idea of completely re-engineering an existing software system with current technology is not feasible. There is, however, a threshold that exists where the expected life span of a software system justifies that the system would be better maintained after being re-engineered in this technology. Total re-engineering should be planned rather than as a reaction to a crisis situation. At minimum, prototyping technology could be used on critical portion of an existing software system. This minimal approach could be used as a means to transition an organization to total re-engineering.

Software prototyping must be integrated within an organization through training, case studies, and library development. In situations where this full range of commitment to this technology is lacking, e.g. only developers training is provided, when problems begin to arise in using the technology a normal reaction of management is to revert back to what has worked in the past.

The end user involvement becomes enhanced when changes in requirements can be prototyped and agreed to before any development proceeds. Similarly, during development of

the actual system or even later into maintenance, should the requirements change; the prototype is enhanced and agreed to before the actual changes become confirmed.

3.4 REQUIREMENTS DOCUMENTATION

Requirements documentation is very important activity after the requirements elicitation and analysis. This is the way to represent requirements in a consistent format. Requirements document is called Software Requirements Specification (SRS).

The SRS is a specification for a particular Software product, program or set of programs that performs certain functions in a specific environment. It serves a number of purposes depending on who is writing it. First, the SRS could be written by the customer of a system. Second, the SRS could be written by a developer of the system. The two scenarios create entirely different situations and establish entirely different purposes for the document. First case, SRS is used to define the needs and expectations of the users. The second case, SRS is written for different purpose and serve as a contract document between customer and developer.

This reduces the probability of the customer being disappointed with the final product. The SRS written by developer (second case) is of our interest and discussed in the subsequent sections.

3.4.1 Nature of the SRS

The basic issues that SRS writer(s) shall address are the following:

1. **Functionality:** What the software is supposed to do?
2. **External Interfaces:** How does the software interact with people, the system's hardware, other hardware, and other software?
3. **Performance:** What is the speed, availability, response time, recovery time, etc. of various software functions?
4. **Attributes:** What are the considerations for portability, correctness, maintainability, security, reliability etc.?
5. **Design constraints imposed on an implementation:** Are there any required standards in effect, implementation language, policies for database integrity, resource limits, operating environment(s) etc.?

Since the SRS has a specific role to play in the software development process, SRS writer(s) should be careful not to go beyond the bounds of that role. This means the SRS

1. should correctly define all the software requirements. A software requirement may exist because of the nature of the task to be solved or because of a special characteristic of the project.
2. should not describe any design or implementation details. These should be described in the design stage of the project.
3. should not impose additional constraints on the software. These are properly specified in other documents such as a software quality assurance plan.

Therefore, a properly written SRS limits the range of valid designs, but does not specify any particular design.

3.4.2 Characteristics of a good SRS

The SRS should be:

Correct
Unambiguous
Complete
Consistent
Ranked for importance and/or stability
Verifiable
Modifiable
Traceable

Each of the above mentioned characteristics is discussed below: [THAY97, IEEE87, IEEE97].

Correct

The SRS is correct if, and only if; every requirement stated therein is one that the software shall meet. There is no tool or procedure that assures correctness. If the software must respond to all button presses within 5 seconds and the SRS stated that "the software shall respond to all buttons presses within 10 second", then that requirement is incorrect.

Unambiguous

The SRS is unambiguous if, and only if; every requirement stated therein has only one interpretation. Each sentence in the SRS should have the unique interpretation. Imagine that a sentence is extracted from the SRS, given to ten people who are asked for their interpretation. If there is more than one such interpretation, then that sentence is probably ambiguous.

In cases, where a term used in a particular context could have multiple meanings, the term should be included in a glossary where its meaning is made more specific. The SRS should be unambiguous to both those who create it and to those who use it. However, these groups often do not have the same background and therefore do *not tend to describe* software requirements in the same way.

Requirements are often written in natural language (for example, English). Natural language is inherently ambiguous. A natural language SRS should be reviewed by an independent party to identify ambiguous use of a language so that it can be corrected. This can be avoided by using a particular requirement specification language. Its language processors automatically detect many lexical, syntactic, and semantic errors. Disadvantage is the time required to learn the language which may also not be understandable to the customers/users. Moreover, these languages tend to be better at expressing only certain types of requirements and addressing certain types of systems.

Complete

The SRS is complete if, and only if; it includes the following elements:

1. All significant requirements, whether relating to functionality, performance, design constraints, attributes or external interfaces.

2. Definition of their responses of the software to all realizable classes *of input data in all realizable classes of situations. Note that it is important to specify the responses to both valid and invalid values.*
3. Full labels and references to all figures, tables, and diagrams in the SRS and definition of all terms and units of measure.

Consistent

The SRS is consistent if, and only if, no subset of individual requirements described in it conflict. There are three types of likely conflicts in the SRS:

1. The specified characteristics of real-world objects may conflict. For example.
   (*a*) The format of an output report may be described in one requirement as tabular but in another as textual.
   (*b*) One requirement may state that all lights shall be green while another states that all lights shall be blue.
2. There may be logical or temporal conflict between two specified actions, for example,
   (*a*) One requirement may specify that the program will add two inputs and another may specify that the program will multiply them.
   (*c*) One requirement may state that "A" must always follow "B", while another requires that "A and B" occur simultaneously.
3. Two or more requirements may describe the same real-world object but use different terms for that object. For example, a program's request for a user input may be called a "prompt" in one requirement and a "cue" in another. The use of standard terminology and definitions promotes consistency.

Ranked for importance and/or stability

The SRS is ranked for importance and/or stability if each requirement in it has an identifier to indicate either the importance or stability of that particular requirement.

Typically, all requirements are not equally important. Some requirements may be essential, especially for life critical applications, while others may be desirable. Each requirement should be identified to make these differences clear and explicit. Another way to rank requirements is to distinguish classes of requirements as essential, conditional and optional.

Verifiable

The SRS is verifiable, if and only if, every requirement stated therein is verifiable. A requirement is verifiable, if and only if, there exists some finite cost-effective process with which a person or machine can check that the software meets the requirements. In general any ambiguous requirement is not verifiable.

Nonverifiable requirements include statement, such as "works well", "good human interface", and "shall usually happen". These requirements cannot be verified because it is impossible to define the terms "good", "well", or "usually". The statement that "the program

shall never enter an infinite loop" is nonverifiable 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.

3.4.3 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 organization of an SRS is given in Fig. 3.18 [IEEE93].

1. Introduction
   - 1.1 Purpose
   - 1.2 Scope
   - 1.3 Definitions, Acronyms, and Abbreviations
   - 1.4 References
   - 1.5 Overview
2. The Overall Description
   - 2.1 Product Perspective
     - 2.1.1 System Interfaces
     - 2.1.2 Interfaces
     - 2.1.3 Hardware Interfaces
     - 2.1.4 Software Interfaces
     - 2.1.5 Communications Interfaces
     - 2.1.6 Memory Constraints
     - 2.1.7 Operations
     - 2.1.8 Site Adaptation Requirements
   - 2.2 Product Functions
   - 2.3 User Characteristics
   - 2.4 Constraints
   - 2.5 Assumptions and Dependencies.
   - 2.6 Apportioning of Requirements
3. Specific Requirements
   - 3.1 External interfaces
   - 3.2 Functions
   - 3.3 Performance Requirements
   - 3.4 Logical Database Requirements
   - 3.5 Design Constraints
     - 3.5.1 Standards Compliance
   - 3.6 Software System Attributes
     - 3.6.1 Reliability
     - 3.6.2 Availability
     - 3.6.3 Security
     - 3.6.4 Maintainability
     - 3.6.5 Portability
   - 3.7 Organizing the Specific Requirements
     - 3.7.1 System Mode
     - 3.7.2 User Class
     - 3.7.3 Objects
     - 3.7.4 Feature
     - 3.7.5 Stimulus
     - 3.7.6 Response
     - 3.7.7 Functional Hierarchy
   - 3.8 Additional Comments
4. Change Management Process
5. Document Approvals
6. Supporting Information

Fig. 3.18: Organisation of SRS [IEEE-std. 830-1993].

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 organization

(*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 organized.

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 organization

(*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 organized 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 organizing 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/organization
- Window formats/organization
- 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 Organizing 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 organizing these in a manner optimal for understanding. There is no one optimal organization for all systems. Different classes of systems lend themselves to different organizations of requirements. Some of these organizations are described in the following subclasses.

3.7.1 System Mode

Some systems behave quite differently depending on the mode of operation. When organizing 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 organized by describing their functions in terms of stimuli.

3.7.6 Response

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

3.7.7 Functional Hierarchy

When none of the above organizational schemes prove helpful, the overall functionality can be organized into a hierarchy of functions organized 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 organizational techniques given in 3.7 may be appropriate. In such cases, organize 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 organization. For example, when organizing by mode, finite state machines or state charts may prove helpful, when organizing by object, object-oriented analysis may prove helpful; when organizing by feature, stimulus-response sequences may prove helpful; when organizing 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 subsections 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.

Outline for SRS Section 3
Organized by Mode: Version 1

3 Specific Requirements
- 3.1 External interface requirements
  - 3.1.1 User interfaces
  - 3.1.2 Hardware interfaces
  - 3.1.3 Software interfaces
  - 3.1.4 Communications interfaces
- 3.2 Functional requirements
  - 3.2.1 Mode 1
    - 3.2.1.1 Functional requirements 1.1 .....
    - 3.2.1.*n* Functional requirements 1.*n*
  - 3.2.2 Mode 2 .....
  - 3.2.*m* Mode *m*
    - 3.2.*m*.1 Functional requirement *m*.1 .....
    - 3.2.*m*.*n* Functional requirement *m*.*n*
- 3.3 Performance Requirements
- 3.4 Design Constraints
- 3.5 Software system attributes
- 3.6 Other requirements

Outline for SRS Section 3
Organized by Mode: Version 2

3 Specific Requirements
- 3.1 Functional Requirements
  - 3.1.1 Mode 1
    - 3.1.1.1 External interfaces
    - 3.1.1.1 User Interfaces
    - 3.1.1.2 Hardware interfaces
    - 3.1.1.3 Software interfaces
    - 3.1.1.4 Communications interfaces
    - 3.1.1.2 Functional Requirement
      - 3.1.1.2.1 Functional requirement 1 .....
      - 3.1.1.2.1.*n* Functional requirement *n*
    - 3.1.1.3 Performance
  - 3.1.2 Mode 2 .....
  - 3.1.*m* Mode *m*
- 3.2 Design constraints
- 3.3 Software system attributes
- 3.4 Other requirements.

Outline for SRS Section 3
Organized by User Class

3 Specific Requirements
- 3.1 External interface requirements
  - 3.1.1 User interfaces
  - 3.1.2 Hardware interfaces
  - 3.1.3 Software interfaces
  - 3.1.4 Communications interfaces
- 3.2 Functional requirements
  - 3.2.1 User class 1
    - 3.2.1.1 Functional requirements 1.1 .....
    - 3.2.1.*n* Functional requirement 1.*n*
  - 3.2.2 User class 2 .....
  - 3.2.*m* User class *m*
    - 3.2.*m*.1 Functional requirement *m*.1 .....
    - 3.2.*m*.*n* Functional requirement *m*.*n*
- 3.3 Performance Requirements
- 3.4 Design Constraints
- 3.5 Software system attributes
- 3.6 Other requirements

Outline for SRS Section 3
Organized by Object

3 Specific Requirements
- 3.1 External interface requirements
  - 3.1.1 User interfaces
  - 3.1.2 Hardware interfaces
  - 3.1.3 Software interfaces
  - 3.1.4 Communications interfaces
- 3.2 Classes/Objects
  - 3.2.1 Class/Object 1
    - 3.2.1.1 Attributes (direct or inherited)
      - 3.2.1.1.1 Attribute 1 .....
      - 3.2.1.1.*n* Attribute *n*
    - 3.2.1.2 Functions (services, methods, direct or inherited)
      - 3.2.1.2.1 Functional requirement 1.1 .....
      - 3.2.1.2.*m* Functional requirement 1.*m*
    - 3.2.1.3 Messages (communications received or sent)
  - 3.2.2 Class/Object 2 .....
  - 3.2.*p* Class/Object *p*
- 3.3 Performance Requirements
- 3.4 Design Constraints
- 3.5 Software system attributes
- 3.6 Other requirements

Outline for SRS Section 3
Organized by Feature

3 Specific Requirements
- 3.1 External interface requirements
  - 3.1.1 User interfaces
  - 3.1.2 Hardware interfaces
  - 3.1.3 Software interfaces
  - 3.1.4 Communications interfaces
- 3.2 System Features
  - 3.2.1 System Feature 1
    - 3.2.1.1 Introduction/Purpose of feature
    - 3.2.1.2 Stimulus/Response sequence
    - 3.2.1.3 Associated functional requirements
      - 3.2.1.3.1 Functional requirement 1 .....
      - 3.2.1.3.*n* Functional requirement *n*
  - 3.2.2 System Feature 2 .....
  - 3.2.*m* System Feature *m* .....
- 3.3 Performance Requirements
- 3.4 Design Constraints
- 3.5 Software system attributes
- 3.6 Other requirements

Outline for SRS Section 3
Organized by Stimulus

3 Specific Requirements
- 3.1 External interface requirements
  - 3.1.1 User interfaces
  - 3.1.2 Hardware interfaces
  - 3.1.3 Software interfaces
  - 3.1.4 Communications interfaces
- 3.2 Functional requirements
  - 3.2.1 Stimulus 1
    - 3.2.1.1 Functional requirement 1.1 .....
    - 3.2.1.*n* Functional requirement 1.*n*
  - 3.2.2 Stimulus 2 .....
  - 3.2.*m* Stimulus *m*
    - 3.2.*m*.1 Functional requirement *m*.1 .....
    - 3.2.*m*.*n* Functional requirement *m*.*n*
- 3.3 Performance Requirements
- 3.4 Design Constraints
- 3.5 Software system attributes
- 3.6 Other requirements

Outline for SRS Section 3
Organized by Response

3 Specific Requirements
- 3.1 External interface requirements
  - 3.1.1 User interfaces
  - 3.1.2 Hardware interfaces
  - 3.1.3 Software interfaces
  - 3.1.4 Communications interfaces
- 3.2 Functional requirements
  - 3.2.1 Response 1
    - 3.2.1.1 Functional requirement 1.1 .....
    - 3.2.1.*n* Functional requirement 1.*n*
  - 3.2.2 Response 2 .....
  - 3.2.*m* Response *m*
    - 3.2.*m*.1 Functional requirement *m*.1.....
    - 3.2.*m*.*n* Functional requirement *m*.*n*
- 3.3 Performance Requirements
- 3.4 Design Constraints
- 3.5 Software system attributes
- 3.6 Other requirements

Outline for SRS Section 3
Organized by Functional Hierarchy

3 Specific Requirements
- 3.1 External interface requirements
  - 3.1.1 User interfaces
  - 3.1.2 Hardware interfaces
  - 3.1.3 Software interfaces
  - 3.1.4 Communications interfaces
- 3.2 Functional requirements
  - 3.2.1 Information flows
    - 3.2.1.1 Data flow diagram 1
      - 3.2.1.1.1 Data entities
      - 3.2.1.1.2 Pertinent processes
      - 3.2.1.1.3 Topology
    - 3.2.1.2 Data flow diagram 2
      - 3.2.1.2.1 Data entities
      - 3.2.1.2.2 Pertinent processes
      - 3.2.1.2.3 Topology .....
    - 3.2.1.*n* Data flow diagram *n*
      - 3.2.1.*n*.1 Data entities
      - 3.2.1.*n*.2 Pertinent processes
      - 3.2.1.*n*.3 Topology

(Contd)...

3.2.2 Process descriptions
- 3.2.2.1 Process 1
  - 3.2.2.1.1 Input data entities
  - 3.2.2.1.2 Algorithm or formula of process
  - 3.2.2.1.3 Affected data entities
- 3.2.2.2 Process 2
  - 3.2.2.2.1 Input data entities
  - 3.2.2.2.2 Algorithm or formula of process
  - 3.2.2.2.3 Affected data entities .....
- 3.2.2.*m* Process *m*
  - 3.2.2.*m*.1 Input data entities
  - 3.2.2.*m*.2 Algorithm or formula of process
  - 3.2.2.*m*.3 Affected data entities

3.2.3 Data construct specifications
- 3.2.3.1 Construct 1
  - 3.2.3.1.1 Record type
  - 3.2.3.1.2 Constituent fields
- 3.2.3.2 Construct 2
  - 3.2.3.2.1 Record type
  - 3.2.3.2.2 Constituent fields .....
- 3.2.3.*p* Construct *p*
  - 3.2.3.*p*.1 Record type
  - 3.2.3.*p*.2 Constituent fields

3.2.4 Data dictionary
- 3.2.4.1 Data element 1
  - 3.2.4.1.1 Name
  - 3.2.4.1.2 Representation
  - 3.2.4.1.3 Units/Format
  - 3.2.4.1.4 Precision/Accuracy
  - 3.2.4.1.5 Range
- 3.2.4.2 Data element 2
  - 3.2.4.2.1 Name
  - 3.2.4.2.2 Representation
  - 3.2.4.2.3 Units/Format
  - 3.2.4.2.4 Precision/Accuracy
  - 3.2.4.2.5 Range .....
- 3.2.4.*q* Data element *q*
  - 3.2.4.*q*.1 Name
  - 3.2.4.*q*.2 Representation
  - 3.2.4.*q*.3 Units/Format
  - 3.2.4.*q*.4 Precision/Accuracy
  - 3.2.4.*q*.5 Range

3.3 Performance Requirements

3.4 Design Constraints

3.5 Software system attributes

3.6 Other requirements

Outline for SRS Section 3
Showing Multiple Organizations

3 Specific Requirements
- 3.1 External interface requirements
  - 3.1.1 User interfaces
  - 3.1.2 Hardware interfaces
  - 3.1.3 Software interfaces
  - 3.1.4 Communications interfaces
- 3.2 Functional requirements
  - 3.2.1 User class 1
    - 3.2.1.1 Feature 1.1
      - 3.2.1.1.1 Introduction/Purpose of feature
      - 3.2.1.1.2 Stimulus/Response sequence
      - 3.2.1.1.3 Associated functional requirements
    - 3.2.1.2 Feature 1.2
      - 3.2.1.2.1 Introduction/Purpose of feature
      - 3.2.1.2.2 Stimulus/Response sequence
      - 3.2.1.2.3 Associated functional requirements .....
    - 3.2.1.*m* Feature 1.*m*
      - 3.2.1.*m*.1 Introduction/Purpose of feature
      - 3.2.1.*m*.2 Stimulus/Response sequence
      - 3.2.1.*m*.3 Associated functional requirements
  - 3.2.2 User class 2 .....
  - 3.2.*n* User class *n* .....
- 3.3 Performance Requirements
- 3.4 Design Constraints
- 3.5 Software system attributes
- 3.6 Other requirements

3.5 STUDENT RESULT MANAGEMENT SYSTEM—EXAMPLE

A university has decided to engage a software company for the automation of student result management system of its M. Tech. Programme. The following documents are required to be prepared.

(*i*) Problem statement
(*ii*) Context diagram
(*iii*) Data flow diagrams
(*iv*) ER diagrams
(*v*) Use case diagram
(*vi*) Use cases
(*vii*) SRS as per IEEE std. 830-1993

These seven documents may provide holistic view of the systme to be developed. The SRS will act as contract document between developers (software company) and client (University).

3.5.1 Problem Statement

The problem statement is the first document which is normally prepared by the client. It only, gives superficial view of the system as per client's perspective and expectations. it is the input to the requirement engineering process where final product is the SRS.

The problem statement of student result management system of M. Tech. (Information Technology) Programme of a University is given below:

"A University conducts a 4-semester M. Tech. (IT) program. The students are offered four theory papers and two Lab papers (practicals) during Ist, IInd and IIIrd semesters. The theory papers offered in these semesters are categorized as either 'Core' or 'Elective'. Core papers do not have an alternative subject, whereas elective papers have two other alternative subjects. Thus, a student can study any subject out of the 3 choices available for an elective paper.

In Ist, IInd and IIIrd semesters, 2 core papers and 2 elective papers are offered to each student. The students are also required to submit a term paper/minor project in IInd and IIIrd semesters each. In IVth semester the students have to give a seminar and submit a dissertation on a topic/subject area of their interest.

The evaluation of each subject is done out of 100 marks. During the semester, minor exams are conducted for each semester. Students are also required to submit assignments as directed by the corresponding faculty and maintain Lab records for practicals. Based on the students' performance in minor exams, assignments, Lab records and their attendance, marks out of 40 are given in each subject and practical paper. These marks out of 40 account for internal evaluation of the students. At the end of each semester major exams are conducted in each subject (theory as well as practical). These exams are evaluated out of 60 marks and account for external evaluation of the students. Thus, the total marks of a student in a subject are obtained by adding the marks obtained in internal and external evaluation.

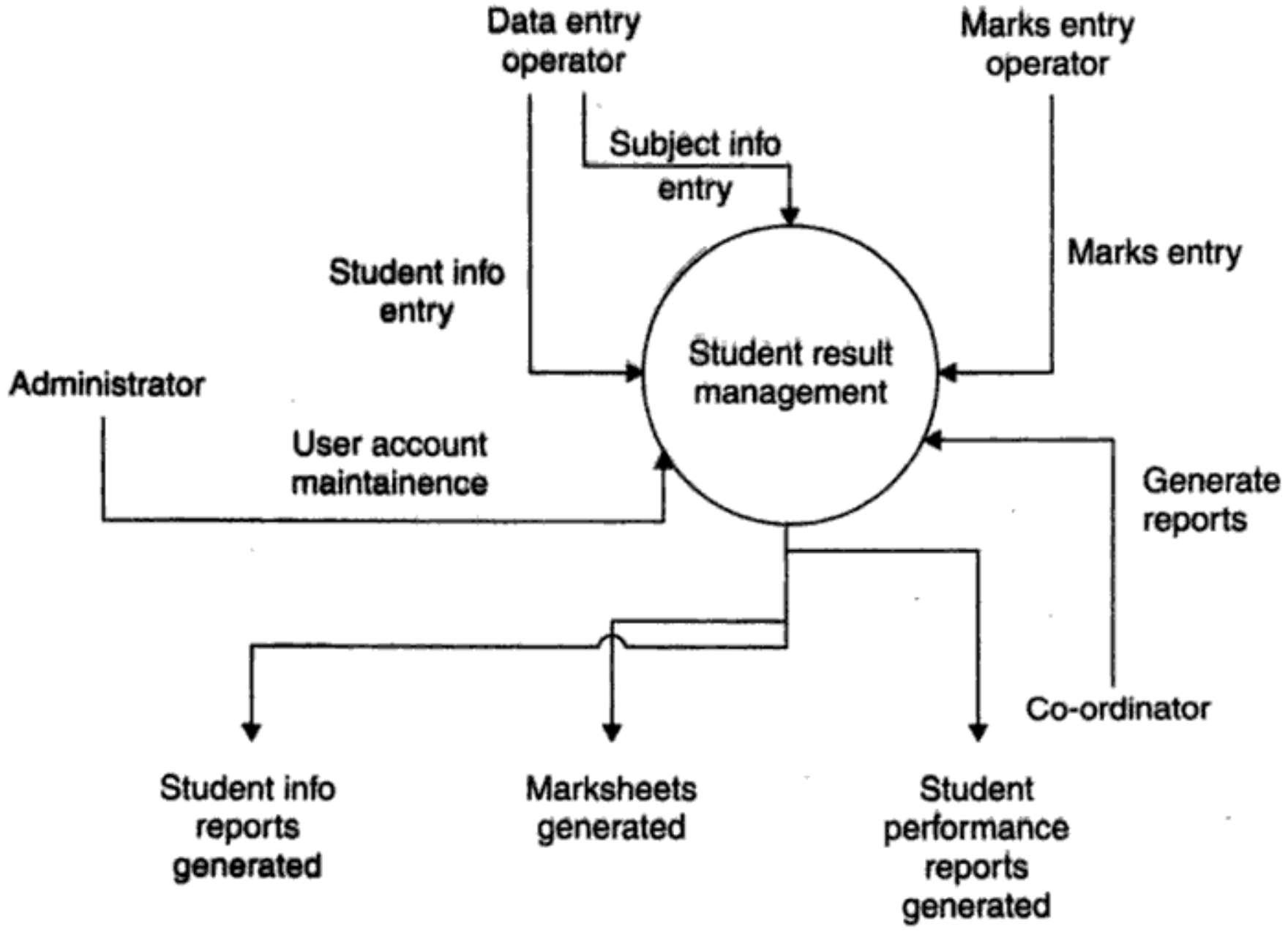
Every subject has some credit points assigned to it. If the total marks of a student are > = 50 in a subject, he/she is considered 'Pass' in that sybject otherwise the student is considered 'Fail' in that subject. If a studnet passes in a subject, he/she earns all the credit points assigned to that subject, but if the students fails in a subject he/she does not earn any credit point in that subject. At any time, the latest information about subjects being offered in various semesters and their credit points can be obtained from university's website.

It is required to develop a system that will manage information about subjects offered in various semesters, students enrolled in various semesters, elective(s) opted by various students in different semesters, marks and credit points obtained by students in different semesters.

The system should also have the ability to generate printable mark sheets for each student. Semester-wise detailed mark lists and student performance reports also need to be generated."

3.5.2 Context Diagram

The context diagram is given below:



The following persons are interacting with the "student result management system"

(*i*) Administrator

(*ii*) Marks entry operator

(*iii*) Data entry operator

(*iv*) Co-ordinator

3.5.3 Level-n DFD

Level-1 DFD

The Level-1 DFD is given below:

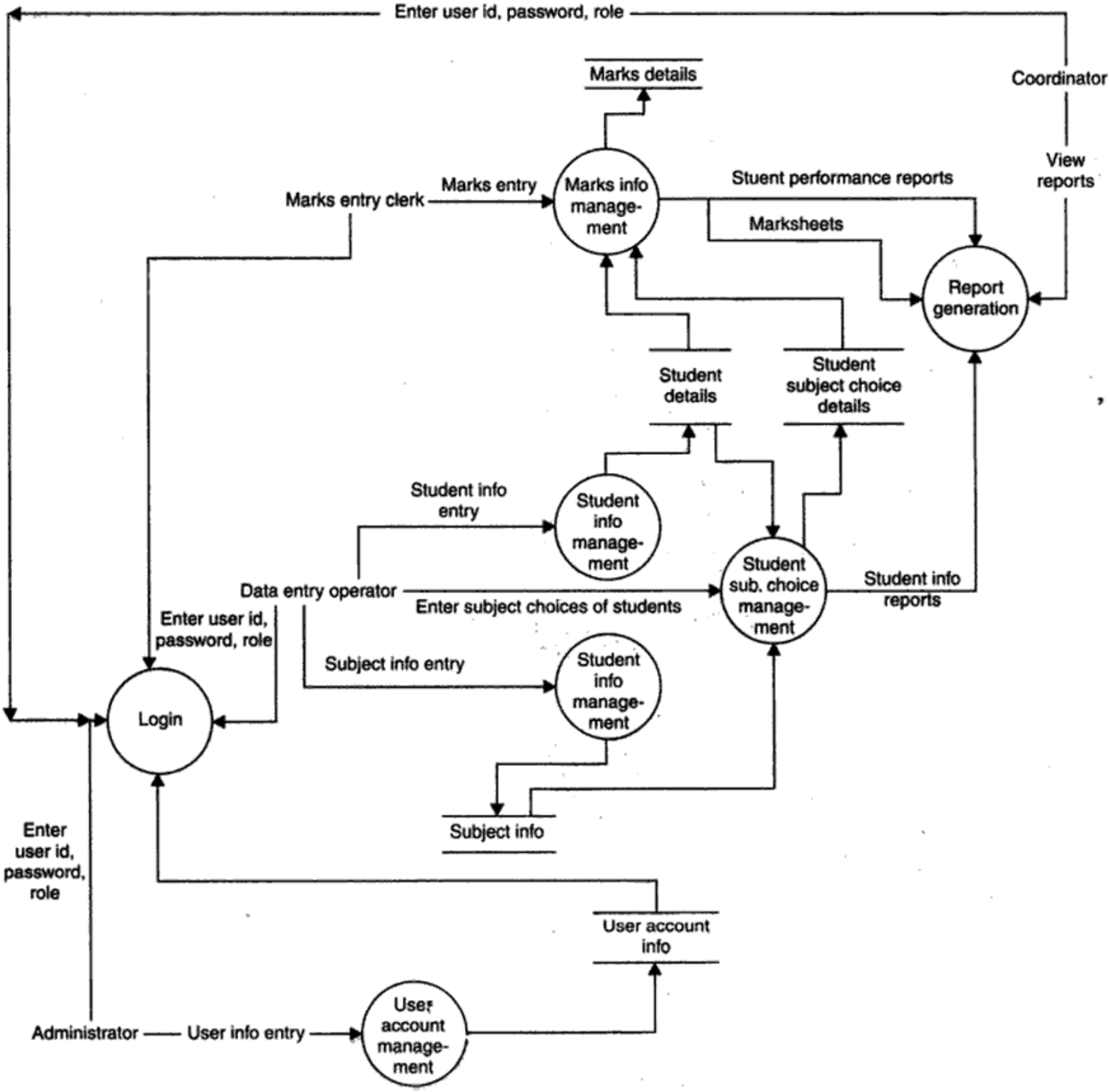
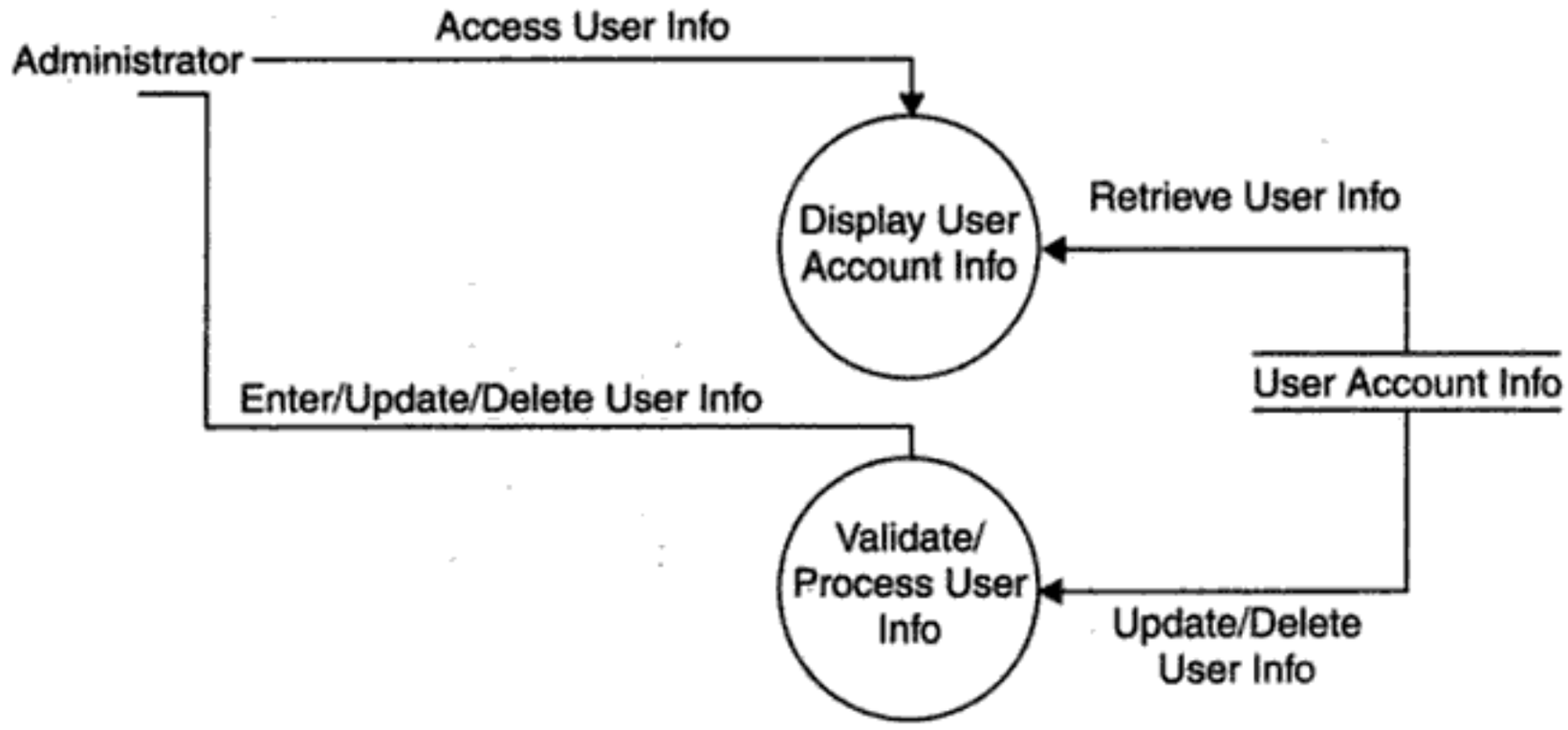


Fig. 3.8: Level 1 DFD of result management system.

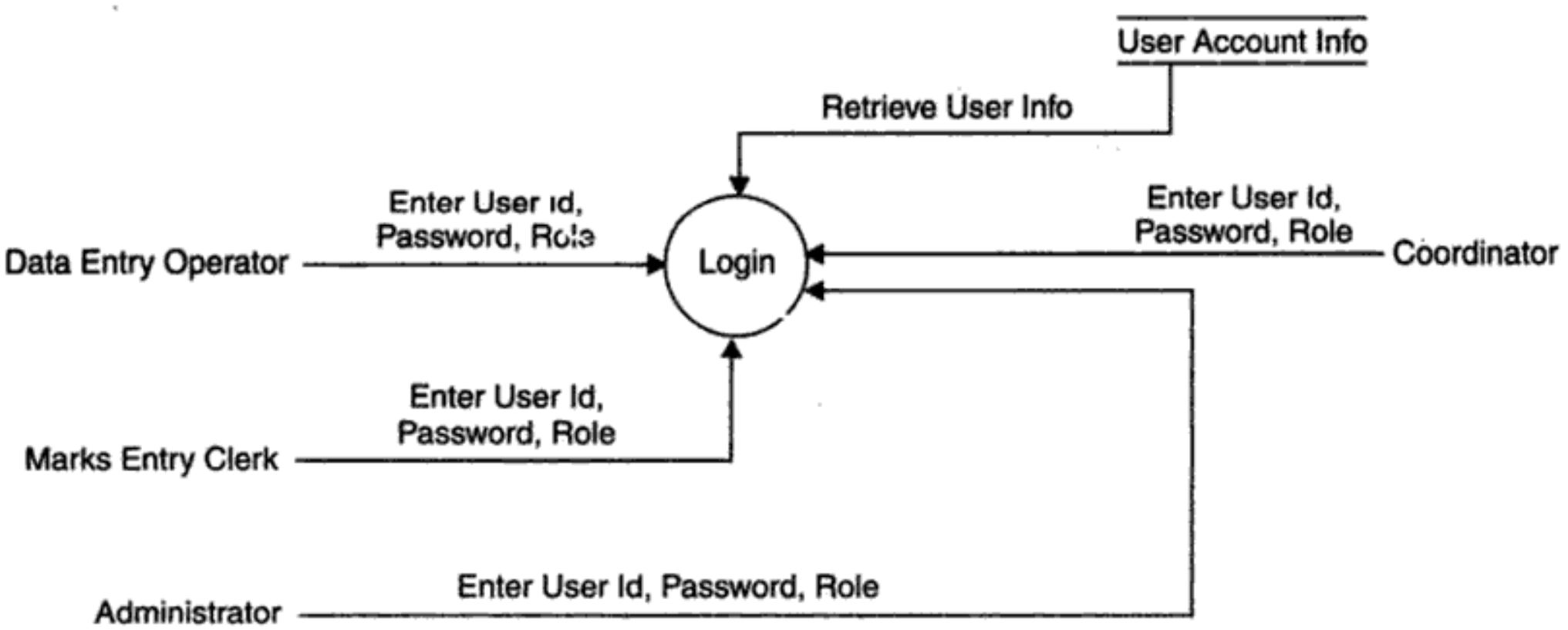
Level-2 DFDs

1. User Account Maintenance



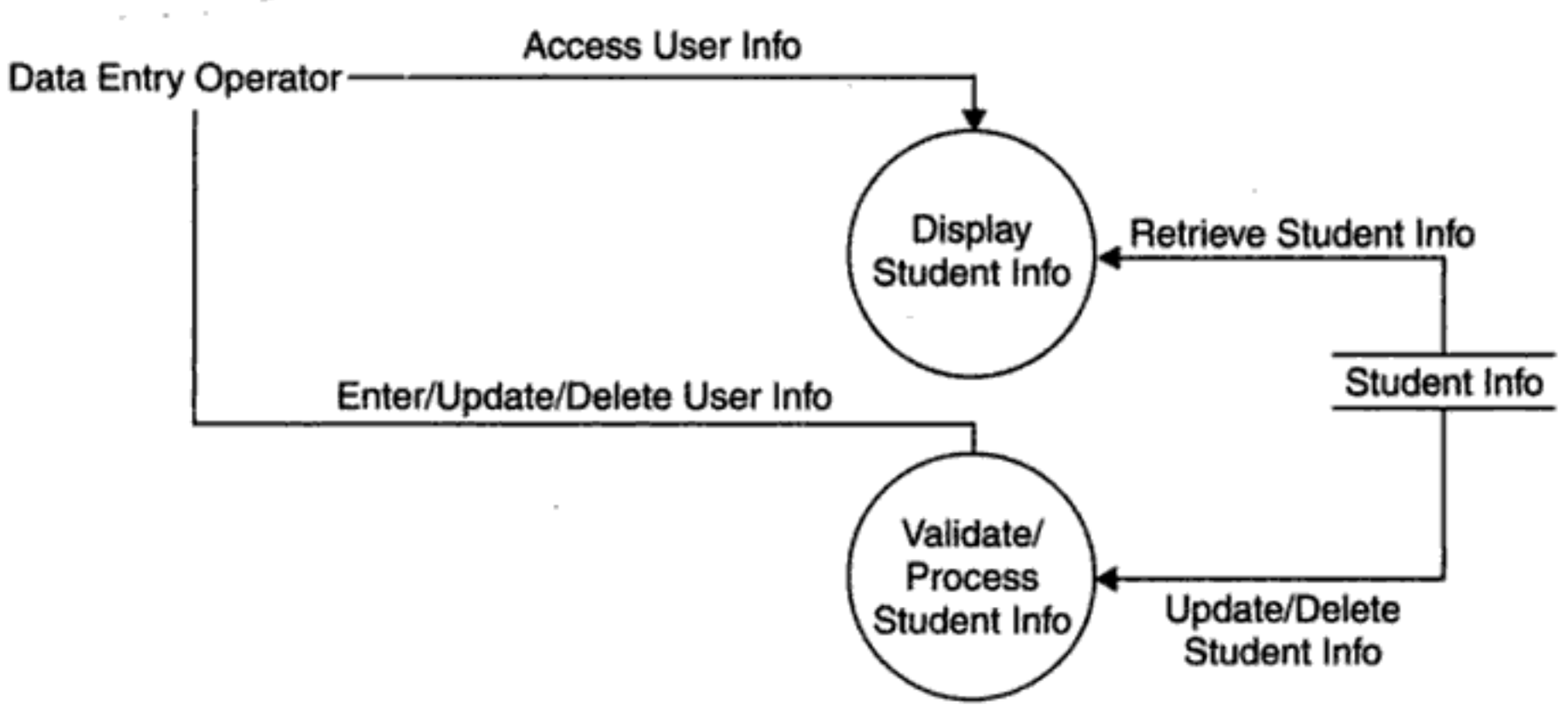
2. Login

The Level 2 DFD of this process is given below:



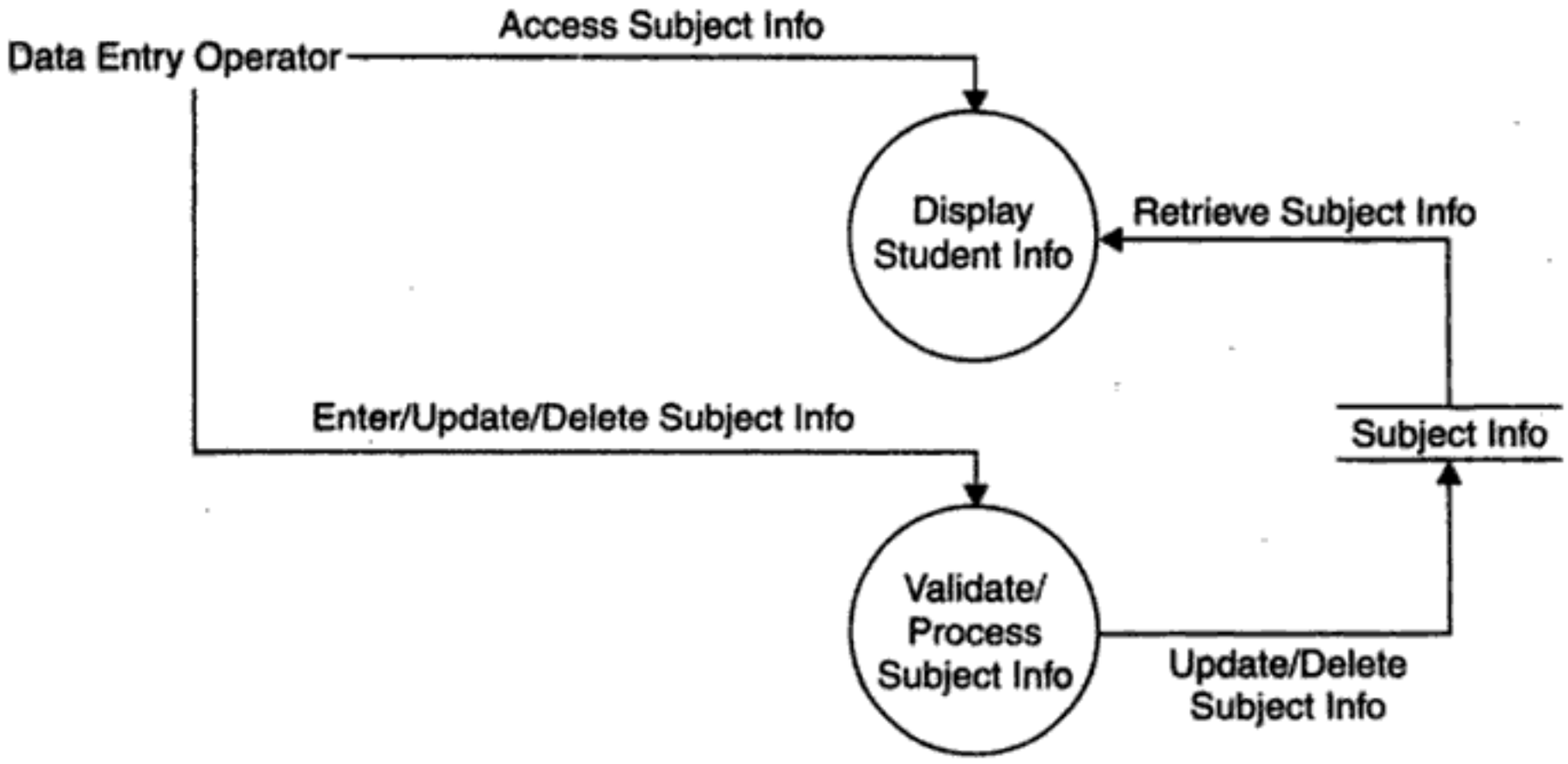
3. Student Information Management

The Level 2 DFD of this process is given below:



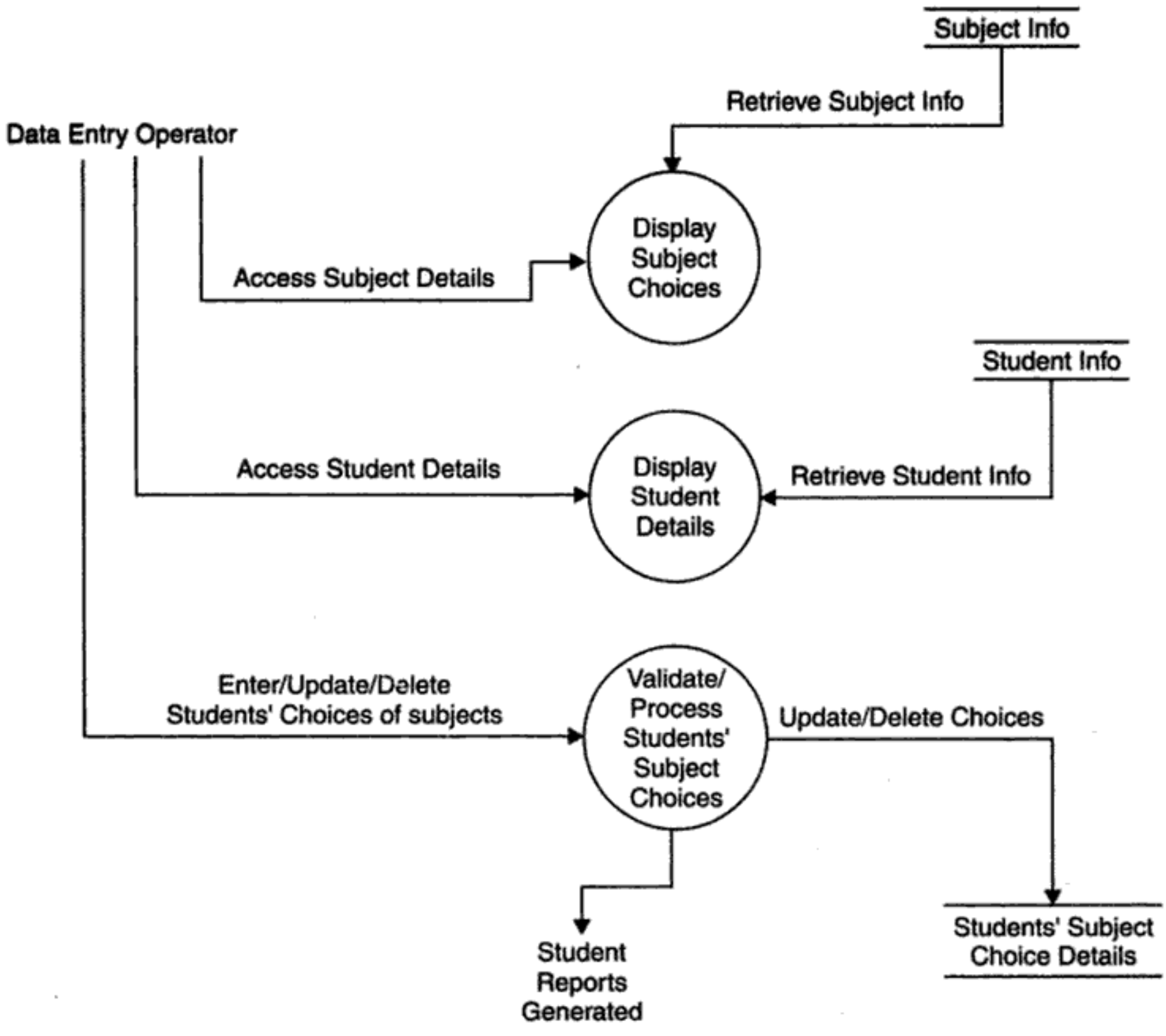
4. Subject Information Management

The Level 2 DFD of this process is given below:



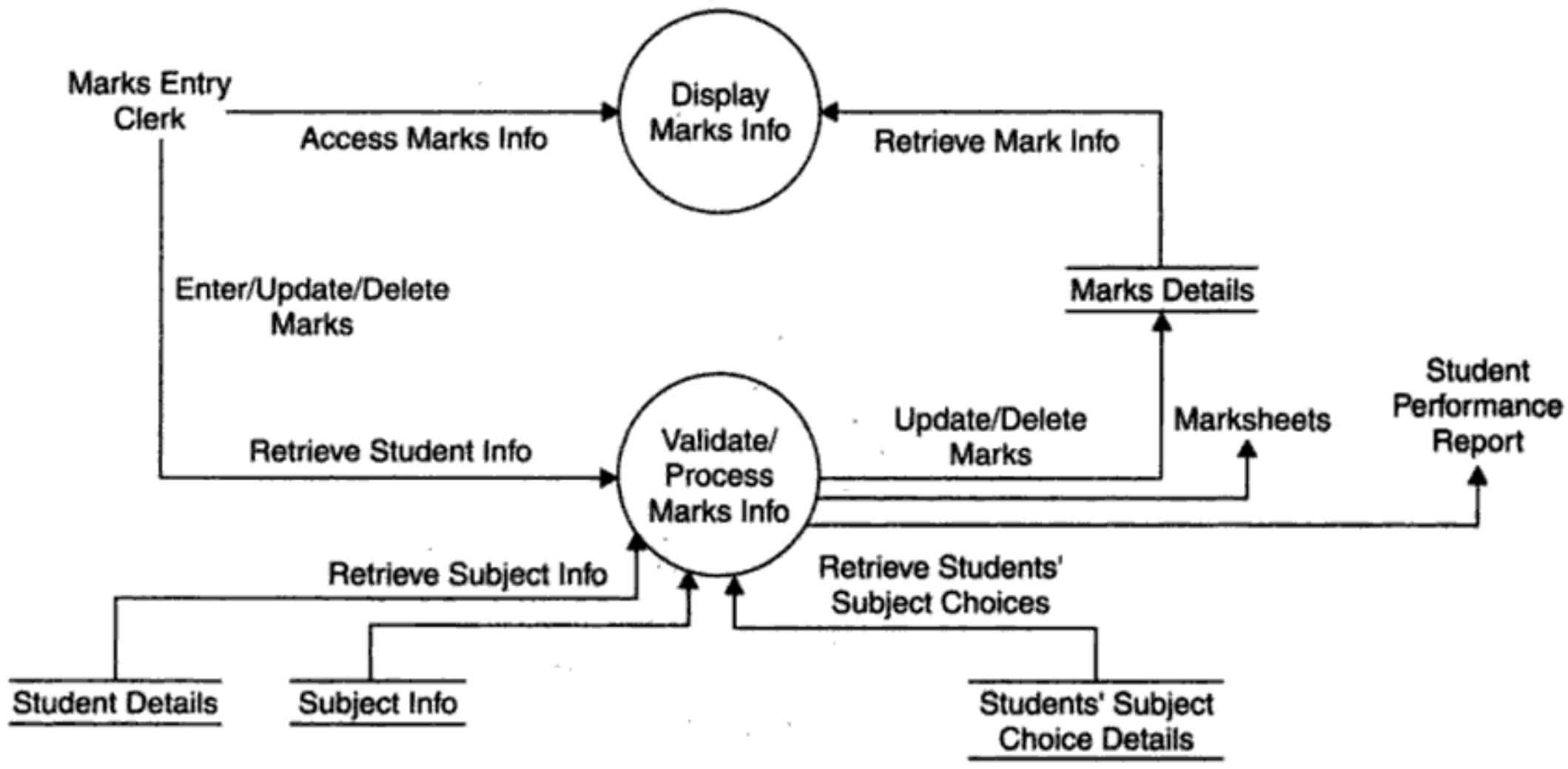
5. Students' Subject Choice Management

The Level 2 DFD of this process is given below:



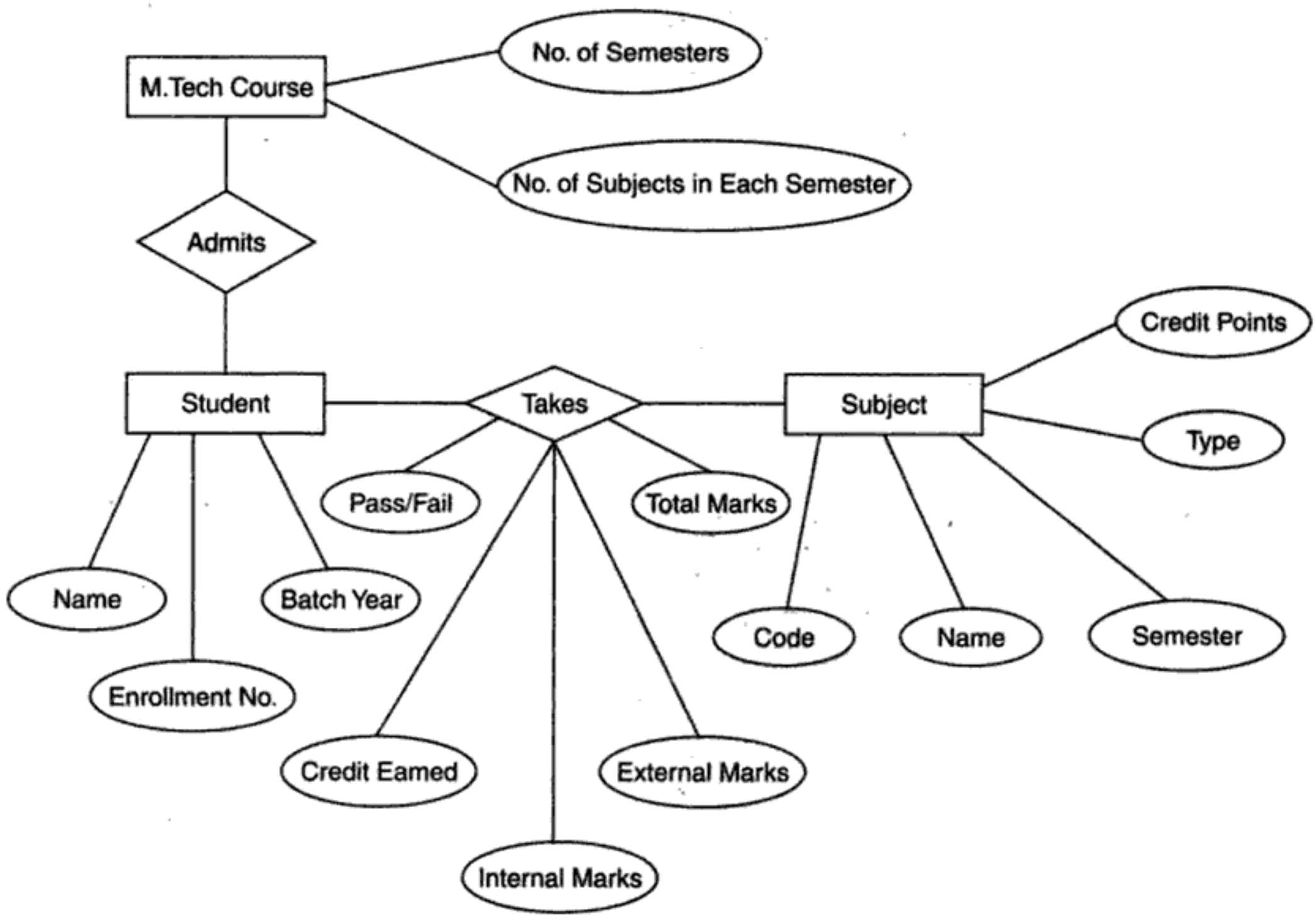
6. Marks Information Management

The Level 2 DFD of this process is given below:

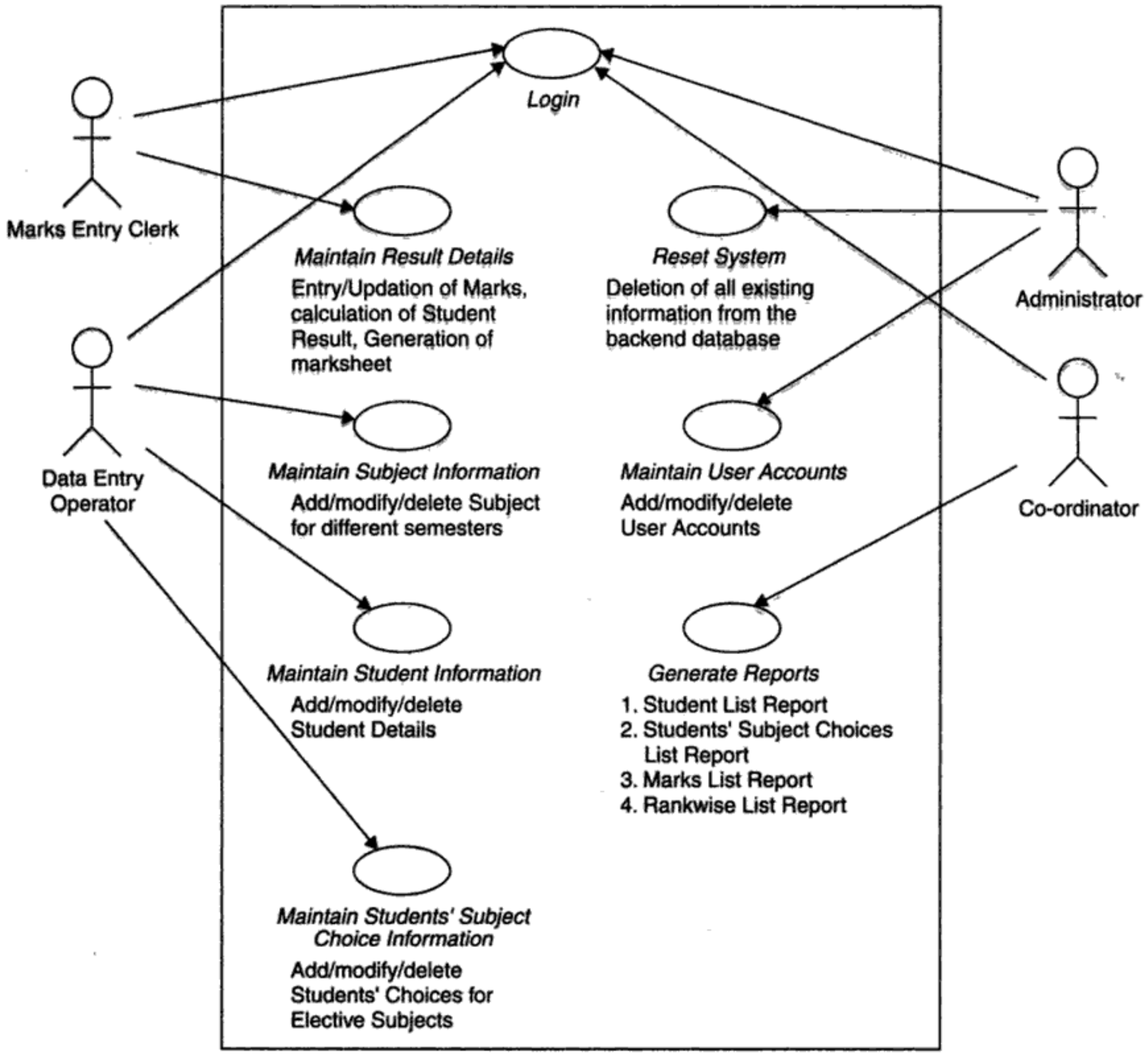


3.5.4 Entity Relationship Diagram

The ER diagram of the system is given below:



3.5.5 Use Case Diagram



Use Case Diagram

3.5.6 Use Cases

1 Login

1.1 Brief Description

This use case describes how a user logs into the Student Result Management System.

1.2 Actors

The following actor(s) interact and participate in this use case:

Data Entry Operator, Marks Entry Clerk, Administrator, Coordinator.

1.3 Flow of Events

1.3.1 Basic Flow

This use case starts when the actor wishes to Login to the Student Result Management System.

1. The system requests that the actor enter his/her name, password and role. The role can be any one of Data Entry Operator, Marks Entry Clerk, Coordinator, and Administrator.
2. The actor enters his/her name, password and role.
3. The system validates the entered name, password, role and logs the actor into the system.

1.3.2 Alternative Flows

1.3.2.1 Invalid Name/Password/Role

If in the Basic Flow, the actor enters an invalid name, password and/or role, the system displays an error message. The actor can choose to either return to the begining of the Basic Flow or cancel the login, at which point the use case ends.

1.4 Special Requirements

None

1.5 Pre-Conditions

All users must have a User Account (*i.e.,* User ID, Password and Role) created for them in the system (through the Administrator), prior to executing the use cases.

1.6 Post-Conditions

If the use case was successful, the actor is logged into the system. If not, the system state is unchanged.

If the actor has the role 'Data Entry Operator' he/she will have access to only screens corresponding to the Subject Info Maintenance, Student Info Maintenance and Students' Subject Choice Info Maintenance modules of the system.

If the actor has the role 'Marks Entrey Clerk', he/she will have access to only screens corresponding to the Marks Info Maintenance module of the system. If the actor has the role 'Coordinator', he/she will only be able to view/print the various reports generated by the system.

If the actor has the role 'Administrator' he/she will have access to only screens corresponding to User Account maintenance module and Reset System feature of the system.

1.7 Extension Points

None

2 Maintain Student Information

2.1 Brief Description

This use case allows the actor with role 'Data Entry Operator' to maintain student information. This includes adding, changing and deleting studnet information from the system.

2.2 Actors

The following actor(*s*) interact and participate in this use case:

Data Entry Operator.

2.3 Flow of Events

2.3.1 Basic Flow

This use case starts when the Data Entry Operator wishes to add, change, and/or delete student information from the system.

1. The system requests that the Data Entry Operator specify the function he/she would like to perform (either Add a Student, Update a Student, or Delete a Student).
2. Once the Data Entry Operator provides the requested information, one of the sub-flows is executed.
   - If the Data Entry Operator selected "Add a Student", the **Add a Student** sub-flow is executed.
   - If the Data Entry Operator selected "Update a Student", the **Update a Student** sub-flow is executed.
   - If the Data Entry Operator selected "Delete a Student", the **Delete a Student** sub-flow is executed.

2.3.1.1 Add a Student

1. The system requests that the Data Entry Operator enter the student information. This includes:
   (*a*) Name
   (*b*) Enrollment Number–should be unique for every student
   (*c*) Year of Enrollment
2. Once the Data Entry Operator provides the requested information, the student is added to the system and an appropriate message is displayed.

2.3.1.2 Update a Student

1. The system requests that the Data Entry Operator enters the student enrollment number.
2. The Data Entry Operator enters the student enrollment number. The system retrieves and displays the student information.
3. The Data Entry Operator makes the desired changes to the student information. This includes any of the information specified in the **Add a Student** sub-flow.
4. Once the Data Entry Operator updates the necessary information, the system updates the student record with the updated information.

2.3.1.3 Delete a Student

1. The system requests that the Data Entry Operator enters the student enrollment number.
2. The Data Entry Operator enters the student enrollment number. The system retrieves and displays the student information.
3. The system prompts the Data Entry Operator to confirm the deletion of the student.
4. The Data Entry Operator confirms the deletion.
5. The system deletes the student record.

2.3.2 Alternative Flows

2.3.2.1 Student Not Found

If in the **Update a Student** or **Delete a Student** sub-flows, a student with the specified enrollment number does not exist, the system displays an error message. The Data Entry Operator can then enter a different enrollment number or cancel the operation, at which point the use case ends.

2.3.2.2 Update Cancelled

If in the **Update a Student** sub-flow, the Data Entry Operator decides not to update the student information, the update is cancelled and the **Basic Flow** is re-started at the beginning.

2.3.2.3 Delete Cancelled

If in the **Delete a Student** sub-flow, the Data Entry Operator decides not to delete the student information, the delete is cancelled and the **Basic Flow** is re-started at the beginning.

2.4 Special Requirements

None

2.5 Pre-Conditions

The Data Entry Operator must be logged onto the system before this use case begins.

2.6 Post-Conditions

If the use case was successful, the student information is added, updated, or deleted from the system. Otherwise, the system state is unchanged.

2.7 Extension Points

None

3 Maintain Subject Information

3.1 Brief Description

This use case allows the actor with role 'Data Entry Operator' to maintain subject information. This includes adding, changing and deleting subject information from the system.

3.2 Actors

The following actor(s) interact and participate in this use case:
Data Entry Operator

3.3 Flow of Events

3.3.1 Basic Flow

This use case starts when the Data Entry Operator wishes to add, change, and/or delete subject information from the system.

1. The system requests that the Data Entry Operator specify the function he/she would like to perform (either Add a Subject, Update a Subject, or Delete a Subject).
2. Once the Data Entry Operator provides the requested information, one of the sub-flows is executed.
   - If the Data Entry Operator selected "Add a Subject", the **Add a Subject** sub-flow is executed.
   - If the Data Entry Operator selected "Update a Subject", the **Update a Subject** sub-flow is executed.
   - If the Data Entry Operator selected "Delete a Subject", the **Delete a Subject** sub-flow is executed.

3.2.1.1 Add a Subject

1. The system requests that the Data Entry Operator enters the subject information. This includes:
   (*a*) Name of the subject
   (*b*) Subject Code–should be unique for every subject
   (*c*) Semester
   (*d*) Subject Type–can be Core 1/Core 2/Dissertation/Elective 1/Elective 2/Lab 1/Lab 2/ Minor Project/Seminar/Term Paper.
   (*e*) Credits.
2. Once the Data Entry Operator provides the requested information, the subject is added to the system and an appropriate message is displayed.

3.3.1.2 Update a Subject

1. The system requests that the Data Entry Operator enters the subject code.
2. The Data Entry Operator enters the subject code. The system retrieves and displays the subject information.
3. The Data Entry Operator makes the desired changes to the subject information. This includes any of the information specified in the **Add a Subject** sub-flow.
4. Once the Data Entry Operator updates the necessary information, the system updates the subject record with the updated information.

3.3.1.3 Delete a Subject

1. The system requests that the Data Entry Operator enter the subject code.
2. The Data Entry Operator enters the subjects code. The system retrieves and displays the subject information.
3. The system prompts the Data Entry operator to confirm the deletion of the subject.
4. The Date Entry Operator confirms the deletion.
5. The system deletes the subject record.

3.3.2 Alternative Flows

3.3.2.1 Subject Not Found

If in the **Update a Subject** or **Delete a Subject** sub-flows, a subject with the specified subject code does not exist, the system displays an error message. The Data Entry Operator can then enter a different subject code or cancel the operation, at which point the use case ends.

3.3.2.2 Update Cancelled

If in the **Update a Subject** sub-flow, the Data Entry Operator decides not to update the subject information, the update is cancelled and the **Basic Flow** is re-started at the beginning.

3.3.2.3 Delete Cancelled

If in the **Delete a Subject** sub-flow, the Data Entry Operator decides not to delete the subject information, the delete is cancelled and the **Basic Flow** is re-started at the beginning.

3.4 Special Requirements

None

3.5 Pre-Conditions

The Data Entry Operator must be logged onto the system before this use case begins.

3.6 Post–Conditions

If the use case was successful, the student information is added, updated, or deleted from the system. Otherwise, the system state is unchanged.

3.7 Extension Points

None

4 Maintain Students' Subject Choice Infromation

4.1 Brief Description

This use case allows the actor with role 'Data Entry Operator' to maintain information about the choice of different Elective subjects opted by various students. This includes displaying the various available choices of Elective subjects available during a particular semester and updating the information about the choice of Elective Subject(s) opted by different students of that semester.

4.2 Actors

The following actor(s) interact and participate in this use case:

Data Entry Operator

4.3 Flow of Events

4.3.1 Basic Flow

This use case starts when the Data Entry Operator wishes to update students' Subject Choice information from the system.

1. The system requests that the Data Entry Operator specify the semester and enrollment year of students, for which the Students' Subject Choices have to be updated.
2. Once the Data Entry Operator provides the requested information, the system displays the list of available choices for Elective I and Elective II subjects for that semester and the list of students enrolled in the given enrollment year (along with their existing subject choices, if any).
3. The system requests that the Data Entry Operator specify the information regarding Students' Subject Choices. this includes
   (*a*) Student's Enrollment Number
   (*b*) Student's Choice for Elective I subject (the corresponding subject code)
   (*c*) Student's Choice for Elective II subject (the corresponding subject code).
4. Once the Data Entry Operator provides the requested information, the information regarding Student's Subject Choices is added/updated in the system and an appropriate message is displayed.

4.3.2 Alternative Flows

4.3.2.1 Subject Information Does Not Exist

If no or incomplete subject information exists in the system for the semester specified by the Data Entry Operator, the system displays an error message. The Data Entry Operator can then enter a different semester or cancel the operation, at which point the use case ends.

4.3.2.2 Student Information Does Not Exist

If no student information exists in the system for the enrollment year specified by the Data entry Operator, the system displays an error message. The Data Entry Operator can then enter a different enrollment year or cancel the operation, at which point the use case ends.

4.3.2.3 Incorrect Choice Entered for Elective I/Elective II Subjects

If the subject code entered by the Data Entry Operator for Elective I/Elective II subject does not exist in the system, the system displays an error message.

The Data Entry Operator can then enter the correct subject code or cancel the operation, at which point the use case ends.

4.3.2.4 Update Cancelled

If in the **Basic Flow**, the Data Entry Operator decides not to update the subject information, the update is cancelled and the **Basic Flow** is re-started at the beginning.

4.4 Special Requirements

None

4.5 Pre-Conditions

The Data Entry Operator must be logged onto the system before this use case begins.

4.6 Post-Conditions

If the use case was successful, information about students' choices for opting different Elective Subjects is added/updated in the system. Otherwise, the system state is unchanged.

4.7 Extension Points

None

5 Maintain Result Details

5.1 Brief Description

This use case allows the actor with role 'Marks Entry Clerk' to maintain subject-wise marks information of each student, in different semesters. This includes adding, changing and deleting marks information from the system.

5.2 Actors

The following actor(s) interact and participate in this use case:
Maks Entry Clerk.

5.3 Flow of Events

5.3.1 Basic Flow

This use case starts when the Marks Entry Clerk wishes to add, change, and/or delete marks information from the system.

1. The system requests that the Marks Entry Clerk specify the function he/she would like to perform (either Add Marks, Update Marks, Delete Marks, or Generate Mark sheet).
2. Once the Marks Entry Clerk provides the requested information, one of the sub-flows is executed.
   - If the Marks Entry Clerk selected "Add Marks", the **Add Marks** sub-flow is executed.
   - If the Marks Entry Clerk selected "Update Marks", the **Update Marks** sub-flow is executed.
   - If the Marks Entry Clerk selected "Delete Marks", the **Delete Marks** sub-flow is executed.
   - If the Marks Entry Clerk selected "Generate Mark sheet", the **Generate Mark sheet** sub-flow is executed.

5.3.1.1 Add Marks Record

1. The system requests that the Marks Entry Clerk enters the marks information. This includes:

(*a*) Selecting a semester

(*b*) Selecting a Subject Code

(*c*) Selecting the student enroliment number

(*d*) Entering the internal/external marks for that semester, subject code and enrollment number.

2. Once the Marks Entry Clerk provides the requested information, the system saves the marks and an appropriate message is displayed.

5.3.1.2 Update Marks Record

1. The system requests the Marks Entry Clerk to make following entries:
   (*a*) Selecting the semester
   (*b*) Selecting the subject code for which marks have to be updated
   (*c*) Selecting the student enrollment number.
2. Once the Marks Entry Clerk provides the requested information, the system retrieves and displays the corresponding marks details.
3. The Marks Entry Clerk makes the desired changes to the internal/external marks details.
4. The system updates the marks record with the changed information.

5.3.1.3 Delete Marks Record

1. The system requests the Marks Entry Clerk to make following entries:
   (*a*) Selecting the semester
   (*b*) Selecting the subject code for which marks have to be updated
   (*c*) Selecting the student enrollment number.
2. Once the Marks Entry Clerk provides the requested information, the system retrievs and displays the corresponding marks record from the database.
3. The system verifies if the Marks Entry Clerk wishes to proceed with the deletion of the record. Upon confirmation, the record is deleted from the system.

5.3.1.4 Compute Result

1. Once all the marks are added to the database, the result is computed for each student.
2. If the student has scored more than 50% in a subject, the associated credit points are allotted to that student.
3. The average percentage marks are claculated for the student and his/her division is also derived based on the percentage.

5.3.1.5 Generate Mark Sheet

1. The system requests that the Marks Entry Clerk specify the Enrollment Number of the student and the semester for which mark sheet is to be generated.
2. Once the Marks Entry Clerk provides the requested information, the system generates a printable mark sheet for the specified student and displays it.
3. The Marks Entry Clerk can then issue a print request for the mark sheet to be printed.

5.3.2 Alternative Flows

5.3.2.1 Record Not Found

If in the **Update Marks, Delete Marks** or **Generate Mark sheet** sub-flows, a record with the specified information does not exist, the system displays an error message. The Marks Entry Clerk can then enter different information for retrieving the record or cancel the operation, at which point the use case ends.

5.3.2.2 Update Cancelled

If in the **Update Marks** sub-flow, the Marks Entry Clerk decides not to update the marks, the update is cancelled and the **Basic Flow** is re-started at the beginning.

5.3.2.3 Delete Cancelled

If in the **Delete Marks** sub-flow, the Marks Entry Clerk decides not to delete the marks, the delete is cancelled and the **Basic Flow** is re-started at the beginning.

5.4 Special Requirements

None

5.5 Pre-Conditions

The Marks Entry Clerk must be logged onto the system before this use case begins.

5.6 Post–Conditions

If the use case was successful, the marks information is added, updated, or deleted from the system. Otherwise, the system state is unchanged.

5.7 Extension Points

None

6 Generate Reports

6.1 Brief Description

This use case allows the actor with role 'Coordinator' to generate various reports. The following reports can be generated:

(*a*) Student List Report
(*b*) Students' Subject Choices List Report
(*c*) Marks List Report
(*d*) Rank-wise List Report

6.2 Actors

The following actor(s) interact and participate in this use case:
Coordinator

SOFTWARE ENGINEERING

Programs • Documentation • Operating Procedures

This book is designed as a textbook for the first course in Software Engineering for undergraduate and postgraduate students. This may also be helpful for software professionals to help them practice the software engineering concepts.

The Second Edition is an attempt to bridge the gap between "What is taught in the classroom" and "What is practiced in the industry". The concepts are discussed with the help of real life examples and numerical problems.

This book explains the basic principles of Software Engineering in a clear and systematic manner. A contemporary approach is adopted throughout the book. After introducing the fundamental concepts, the book presents a detailed discussion of software requirements analysis & specifications. Various norms and models of Software Project Planning are discussed next, followed by a comprehensive account of Software Metrics.

Suitable examples, illustrations, exercises, multiple choice questions and answers are included throughout the book to facilitate an easier understanding of the subject.

Prof. K.K. Aggarwal is the Vice-Chancellor of Guru Gobind Singh Indraprastha University, Delhi. He graduated in Electronics and Communication Engineering from Punjab University and obtained Masters degree in Advanced Electronics from Kurukshetra University securing *First position in both*. Later, he did his Ph.D. in Reliability Evaluation and Optimization also from Kurukshetra University. In 1975, he rose to the level of *Professor at an age of 27½ years*, probably the youngest person in the world to have achieved this level. He has been the Chairman of the Department of Electronics, Communications & Computer Engineering at Regional Engineering College, Kurukshetra for a long time and worked as Dean (Academic) for that institution. He was also Director, "Centre for Excellence for Manpower Development in Reliability Engineering" established by the Ministry of Human Resource Development at that College. Before taking up the *present assignment* in December 1998, Prof. Aggarwal was *Pro Vice-Chancellor*, Guru Jambheshwar University (Technical University of Haryana), Hisar *for a period of three years*.

He has been *President of the Institution of Electronics and Telecommunication Engineers (IETE)* for the period 2002-2004.

Prof. Yogesh Singh is a Professor & the Dean of University School of Information Technology and also the Dean of University School of Engineering & Technology, Guru Gobind Singh Indraprastha University, Delhi. He received his M.Tech. and Ph.D. (Computer Engineering) degrees from National Institute of Technology, Kurukshetra (previously known as Regional Engineering College, Kurukshetra. Prior to this, he was Founder Chairman, Department of Computer Science & Engineering, Guru Jambheshwar University, Hisar, Haryana. His area of research is Software Engineering focusing on Planning, Testing, Metrics and Neural Networks. He has more than 150 publications in International/National Journals and Conferences.

He was the Principal Investigator of the successfully implemented MHRD-AICTE Project entitled "*Experimentation & Development of Software Reliability & Complexity Measurement Techniques*". He is a member of IT-Task force and a member of its Core-group on E-Education, Govt. of NCT of Delhi.

ISBN 81-224-1638-1

PUBLISHING FOR ONE WORLD

NEW AGE INTERNATIONAL (P) LIMITED, PUBLISHERS
New Delhi • Bangalore • Chennai • Cochin • Guwahati • Hyderabad
Jalandhar • Kolkata • Lucknow • Mumbai • Ranchi
www.newagepublishers.com