CHAPTER 9

INFORMATION SYSTEMS AND SOFTWARE ENGINEERING

Syllabus: Information gathering, Requirement and feasibility analysis, Data flow diagrams, Process specifications, Input/output design, Process life cycle, Planning and managing the project, Design, Coding, Testing, Implementation, Maintenance.

9.1 INTRODUCTION

Software engineering is the study of design, development and maintenance of software. Real-time problems are so large and complex that they cannot be solved by single developer. Software engineering defines teams and its quality. It is concerned about all the aspects of software development from feasibility of software to its maintenance. The team in software engineering consists of developers, testers, designers, customers, managers, etc. Every team member works to fulfil task assigned to him.

9.2 INFORMATION SYSTEMS

Information system consists of components such as storing and processing data. All business firms and organisations rely on them to carry out and manage their operations, interact with their customers and suppliers, and compete in the market. For example, these organisations could reach their customers through Internet for the processing of financial accounts and manage their human resources. Other users can rely on information system through online services such as social networking, auctions,

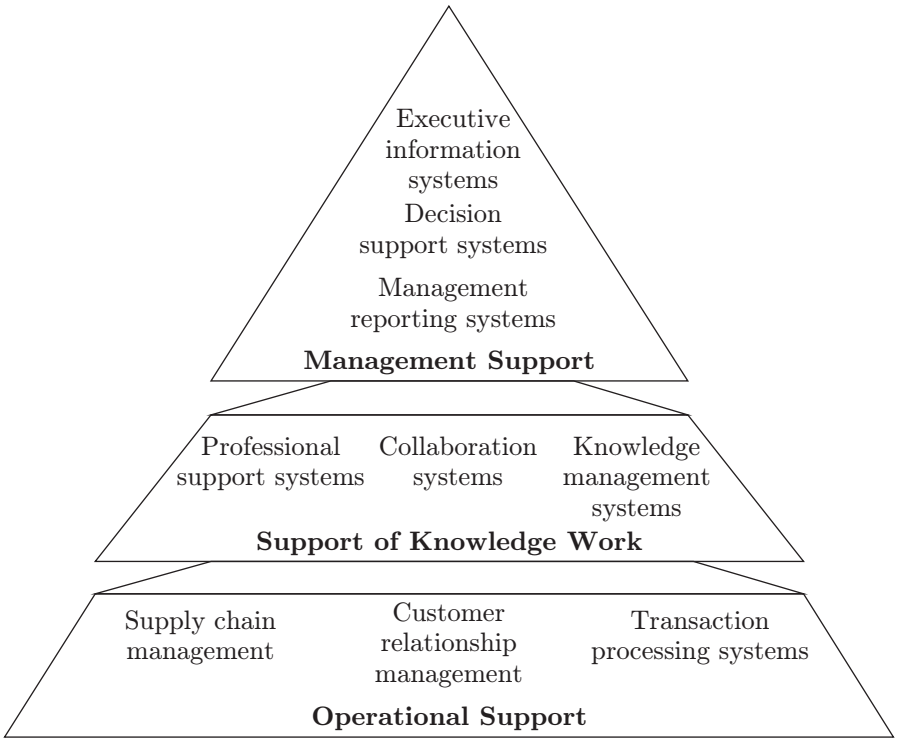


Figure 9.1 An information system.

banking, entertainment, shopping, etc. Information and knowledge have become vital economic resources. Information systems support operations, knowledge work and management in organisations (Fig. 9.1).

9.2.1 Components of Information Systems

The main components of information systems are computer hardware and software, telecommunications, databases and data warehouses, human resources and procedures (Fig. 9.2). The hardware, software and telecommunications constitute information technology (IT), which is now ingrained in the operations and management of organisations.

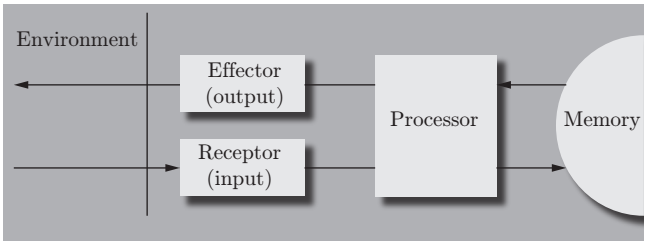


Figure 9.2 Structure of an information system.

9.2.2 Types of Information Systems

9.2.2.1 Office Information System

Office information system (OIS) uses hardware, software and network to enhance the workflow and facilitate communications among employees. OIS is also known as office automation. With the help of OIS, employees or users can perform tasks electronically using computers and other electronic devices, instead of manually processing data. For example, a registration department can post the class schedule via email to students, whereas in a manual system, the registration department would have to photocopy the schedule and mail it to each student's house.

OIS supports a large range of business office activities such as creating and distributing graphics and/or documents, sending messages, scheduling and accounting. All levels of users utilise and benefit from the features of an OIS.

The software used in OIS includes word processing, spreadsheets, databases, presentation graphics, email, Web browsers, Web-page authoring, personal information management and groupware. OIS uses communication technology such as voice mail, fax, video conferencing and electronic data interchange (EDI) for the electronic exchange of text, graphics, audio and video. The hardware includes computers, modems, video cameras, speakers and microphones, scanners and fax machines.

9.2.2.2 Transaction Processing Systems

A transaction processing system (TPS) captures and processes data generated during an organisation's day-to-day

transactions. A transaction is a business activity such as a deposit, payment, order, reservation, etc.

Transaction processing includes the following activities:

1. Recording a business activity such as registration, order, payment, etc.
2. Confirming an action or triggering a response such as printing documents, sending a response, generating pay cheque, issuing receipts, etc.
3. Maintaining data such as adding new data, modifying data or deleting data.

For the processing of business data, TPS was the first computerised system developed as a function called data processing. It is an existing manual system, which allows faster processing, reduced clerical costs and improved customer service.

Online transaction processing (OLTP) is an example of TPS. The administrative users use OLTP to enter data and take print out of the entered data in the form of receipts.

9.2.2.3 Management Information Systems

Management information system (MIS) generates accurate, timely and organised information, and based on that user can make decisions, solve problems, supervise activities and track progress. MIS generates reports on a regular basis, a management information system called management reporting system (MRS). MIS interacts with TPS. On the basis of activities performed by TPS, MIS generates reports of daily activities. The three basic types of information it produces is as follows:

1. **Detailed information:** It confirms transaction processing activities, for example, detailed order report.
2. **Summary information:** It consolidates data into a format, which can be reviewed quickly and easily by a user, for example, an inventory summary report, which includes totals, tables or graphs.
3. **Exception information:** It filters data to report information, which is outside of a normal condition called the exception criteria. An exception criterion is defined as the range of what is considered normal activity or status; for example, exception report, which notifies the items it needs to reorder. It helps in saving time.

9.2.2.4 Decision Support Systems

Decision support system (DSS) is a computer-based information system designed to support decision-making activities such as business or organisation. A variety of DSSs exist having support of wide range of decisions. It uses data from two sources: internal and external.

1. **Internal sources of data:** A data from an organisation such as manufacturing, inventory, sales or financial data.
2. **External source of data:** A data from other organisations such as population trends, interest rates, construction rates, price of raw material, etc.

DSS may also include query language, statistical analysis capabilities, spreadsheets and graphics for the extraction and evaluation of data, if it is very large. DSS includes some kind of capabilities for the creation of a model of the factors affecting the decision. In these models the user can change the values of all or few parameters and project the results. There are lots of application software packages which are used for the decision-making process such as executive information system (EIS), which supports the information needs of executive management. EIS uses data in the form of charts or tabular forms to show trends, ratios and other managerial statistics.

9.2.2.5 Expert Systems

An **expert system** is an information system, which is responsible for capturing and storing human knowledge and imitating human reasoning and decision-making processes. It consists of two main components: knowledge base and inference rules.

1. **Knowledge base:** It is a combination of subject knowledge and experiences of human experts.
2. **Inference rules:** It is a set of logical judgement applied to knowledge base when a user describes a situation to the expert system.

It is used in decision-making at any level in an organisation. It is also used to resolve situations such as diagnosing illnesses, searching for oil, making soup, etc.

It is a part of an artificial intelligence. Artificial intelligence (AI) is an application of human intelligence to computers. It senses human actions based on logical assumptions and prior experience, then take appropriate action to complete the task. It can perform various functions such as speech recognition, logical reasoning and creative responses.

9.2.2.6 Integrated Information Systems

As technology widens its horizons, it becomes difficult to classify a system belonging uniquely to one of the above five types of information system. Today's applications supporting processing of transactions, management of information and decision-making need an integrated information system.

Types of Information Systems

The following are six major types of information systems corresponding to each organisational level (the four levels shown in Fig. 9.1):

1. Transaction processing systems (TPS) to serve the operational level of an organisation.
2. Knowledge work systems (KWS) to keep an organization up-to-date in knowledge in terms of technology, science, art and social thoughts.
3. Office automation systems (OAS) to serve the knowledge level of an organisation.
4. Decision-support systems (DSS) to analyze the existing information to forecast the effects of their decisions in the near future.
5. Management information systems (MIS) to serve the management level of the organisation.
6. Executive support systems (ESS) to serve the strategic level of an organisation.

9.3 SOFTWARE

Software is a program, and when it is executed the desired functionality and performance must be satisfied. Software is a data structure used to manipulate the information. It is a document that describes the operation and use of a program. Finally, it is a logical entity rather than physical entity.

9.3.1 Characteristics of Software

The following are the characteristics of a software:

1. Software is developed or engineered, but not manufactured in classical sense. Although the development approach for both the hardware design and software design is same, but after implementation of the hardware design we get the physical component and after implementation of the software design we get the logical component. In both cases quality is an important factor, that means when the output is operational the associated functionality must be satisfied.
2. Software does not wear out, but deteriorates. In hardware design, at early development, the failure rate is high due to undiscovered errors. After correcting the defects, the failure rate is decreased and the component starts running at a steady state. However, over a period of time, due to external conditions, such as temperature, air, dust and environmental maladies, the failure rate again increases and the hardware starts to wear out. When it undergoes wear out, we can replace the component with a new one. In software design, at early development, the failure rate is high due to undiscovered errors. After correcting the defects the failure rate is decreased and the software starts running within the constrained level. After a period of time, another change is injected in the existing software due to which the failure rate again increases, and after correction the failure rate decreases. So, during the lifetime of software, when the requirements keep changing, the software undergoes deterioration (Fig. 9.3).
3. The industry is moving towards component-based development, but software is still being custom-built. As components are error-free codes, they are reusable. With component-based development for developing some hardware, time and development cost are reduced as the design consists of IC numbers which are already available. So, at the time of implementation we can directly use them. For softwares also, we should use reusable components.

Software development also uses component names such as:

1. **Off the shelf:** This component is used in the project from the third party, that is, the component is not available in the library of the developing organisation.
2. **Fully experienced and partially experienced:** This component is derived based on the previous project, so can be directly used in the current project.
3. **New component:** This component is developed from the base line, means from the initial stage or scratch.

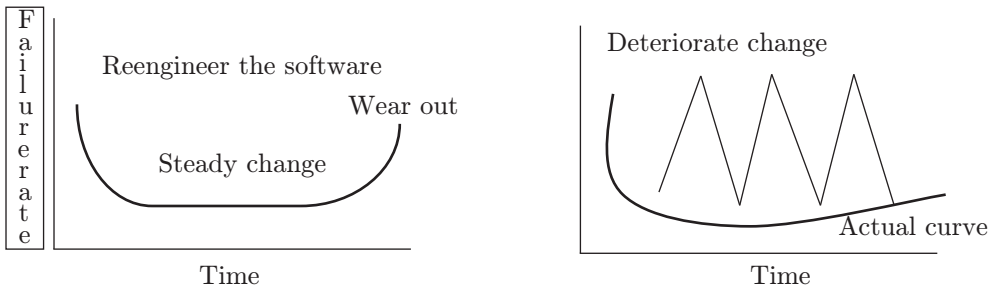


Figure 9.3 | Wear out and deterioration of software.

9.3.2 Software Engineering

Software engineering is the systematic, disciplined and quantitative approach for the development, operation and maintenance of the software.

Software engineering is described in the following two approaches:

1. **Layered approach:** The layered approach consists of the following four layers (Fig. 9.4):
   - ***Quality:*** Confirmation to explicitly stated form as mentioned in the document.
   - ***KPA:*** Key process areas such as planning, schedule, performance and quality attribute.
   - ***Methods:*** "How to" process model.
   - ***Tools:*** Automated and semi-automated tool available to develop the software.

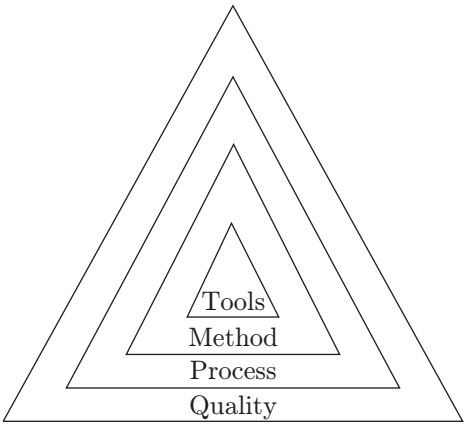


Figure 9.4 Layered approach of software engineering.

2. **Generic approach:** The generic approach consists of the following three stages:
   - ***Definition:*** This stage is focused on 'What', that is, what information is processed, what functionality and performance is desired, what system behaviour is expected, what constraints are imposed and what validation criteria is used. In this stage, analysis and requirement operations are performed.
   - ***Validation:*** 'Are we building the right product?'. The right condition is injected in the testing phase of the development to cover the errors.
   - ***Development:*** This stage is focused on 'how', that is, how to process the information, how to define the data structure, how to maintain the interface, how to convert procedural description to machine readable code, and how to develop test case etc. This case performs design, coding and testing operations.

9.3.2.1 Support

After delivering, the project support stage is required to maintain the software. The following four types of supports or maintenance are required:

1. **Corrective support:** In this support, unidentified errors are going to be managed. The corresponding maintenance is called as corrective maintenance.
2. **Adaptive support:** Over a period of time, if the customer wants to change the platform, that is, CPU, memory, operating system or external interfaces, there is a need of adaptive maintenance.
3. **Perfective (enhancement) support:** Over a period of time, if the customer or the end user wants to introduce new functionality into the existing software to maintain the new software, perfective maintenance is required.
4. **Preventive support:** When a customer keeps on changing the requirement, the software undergoes deterioration. To maintain such a case, there is a need of preventive maintenance or re-engineering.

9.3.2.2 Software Process

A process gives the framework to develop efficient software. On the basis of the process used to develop the software, the organisation undergoes assignment to assess if there is a need of a maturity model.

The Software Engineering Institute (SEI) has introduced a maturity model to assess an organisation. This model is called as the capability maturity model (CMM). The CMM consists of the following five (5) maturity levels:

1. **Level 0 (Initial):** In this level, there is no standard maintained to develop the software. This level uses the adhoc procedure, and development becomes chaotic. Success depends on the individual levels.
2. **Level 1 (Repeatable):** In this level, based on the knowledge of previous project management, activities are developed to trace the cost, schedule and performance. There is no guarantee to the success scenario of the current project.
3. **Level 2 (Defined):** In this level, project management and engineering activities are defined. All the standards are there in the diagrammatical approach, they are not prepared for anticipated situation.
4. **Level 3 (Managed):** In this level, project management and engineering activities are quantitatively collected and documented. Quality attributes are also defined. Because of the lack of continuous improvement, minimum quality is assured.
5. **Level 4 (Optimised):** In this level, more profit is gained with minimum effort while maintaining the continuous improvement process model, innovative ideas, tools and techniques, and qualitative feedback from the customers.

9.4 PROCESS MODELS

There are two types of process models to develop the software (Fig. 9.5).

9.4.1 Conventional Process Model

In order to develop a software, a team of software engineers follow a development strategy specifying the process, methodology, tools and phases. This is referred to as process model. The different types of software models are as follows:

9.4.1.1 Waterfall Model

Waterfall model, also known as linear sequential model, provides a systematic and sequential approach for the development of a software starting from analysis phase to designing, coding, testing and support (Fig. 9.6).

1. **Requirement gathering:** On the basis of the business objective, the input domain and output domain are defined. According to the IEEE standard, requirement is defined as a condition or a capability required by the user to interact with the system or "A documented representation of conditional capability is called as requirement".
2. **Design:** In this stage, the following four operations are performed:
   - *Creating the data structure:* It identifies the data objects and attributes and creates the relationship between the data objects.
   - *Creating the software architecture:* It creates the blue print, that is, the skeleton to represent the inflows and outflows of the software.
   - *Identifying the interfaces:* It represents the interconnection between different modules present in the software.
   - *Defining procedural details:* It converts the high-level abstraction into low-level abstraction (algorithmic approach).
3. **Coding:** It translates the procedural details into machine readable form.
4. **Testing:** In this stage, different test cases are implemented to cover the logical errors and functional errors.
5. **Supports:** After delivering the software to the customer, support stage is involved to maintain the software and to adapt to the changes.

Note: When the requirements are clear, use this model to develop the software. But the final product will be available after a long time schedule.

9.4.1.2 Prototype Model

When customer requirements are not clear, the prototype model is used to finalise the requirement. The framework of prototype model is shown in Fig. 9.7.

In this model, listening to the customer is the entry point to prepare the software requirement specification (SRS). On the basis of the SRS, the designer performs the quick design, quick code. Quick test mock up. The prototype model is evaluated by the customer, and based on the feedback of the customer its iteration continues until the customer is satisfied. After finalising the SRS, the same process model is used to build the final product, which is called as open-end prototyping or evaluating prototyping model. After finalising the SRS, the developer can choose another process model to build the final product, which is called as throwaway prototyping or close-end prototyping.

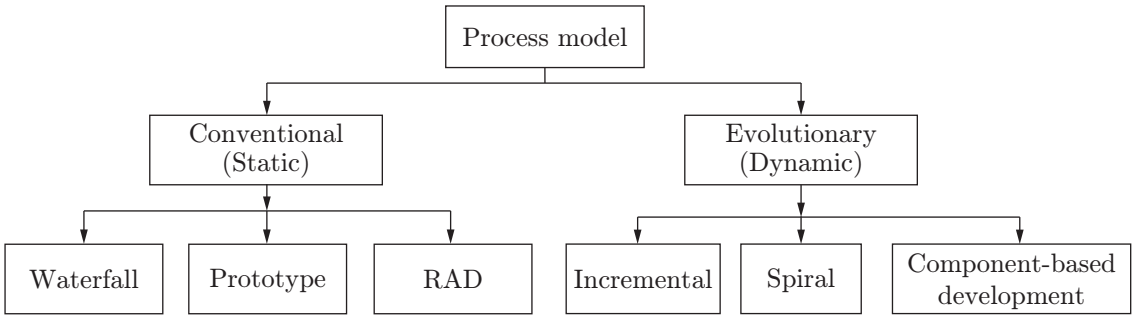


Figure 9.5 Classification of process model.

Figure 9.6 Waterfall model.

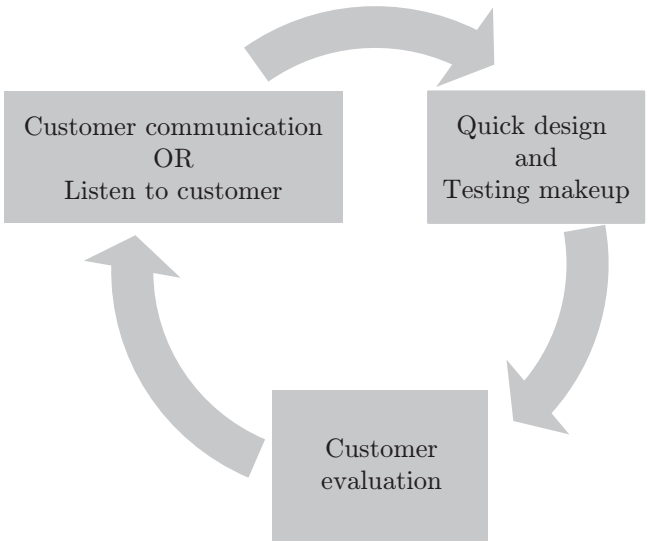


Figure 9.7 | Prototype model.

9.4.1.3 Rapid Application Development

The rapid application development (RAD) model is used when the requirements are clear but the time schedule is very short. In this model, the project is divided into modules and develops the model simultaneously. To maintain effective modularity make sure that the model has high cohesion and low coupling.

1. **Coupling:** It is a quantitative measure of functional strength of a module.
2. **Cohesion:** It is a quantitative measure of to what extent the module is independent from another module.

Framework of the RAD model is shown in Fig. 9.8.

1. **Business modelling:** It identifies the business objective. On the basis of business objective, we can gather data objects.
2. **Data modelling:** It creates the data structure, that is, identifies the data objects and attributes and relationship.
3. **Process modelling:** It creates the information flow in the project by modelling, adding or deleting the data object.
4. **Code generation:** By using a fourth-generation technique, it translates the procedural details into machine-readable format.

Test case: Different test cases are developed and implemented to cover the error. At the end of the test state, all the modules are available with error-free code, which can be integrated to build the final product. RAD model is suitable only when the organisation has efficient manpower.

9.4.2 Evolutionary Process Model

In the evolutionary process model, the software is developed in version basis because requirements are not static. When requirement changes, that change is reflected in the version. Under this case, different models are used to develop effective software.

9.4.2.1 Incremental Model

In this model, classic life cycle is used to develop the software. See Fig. 9.9.

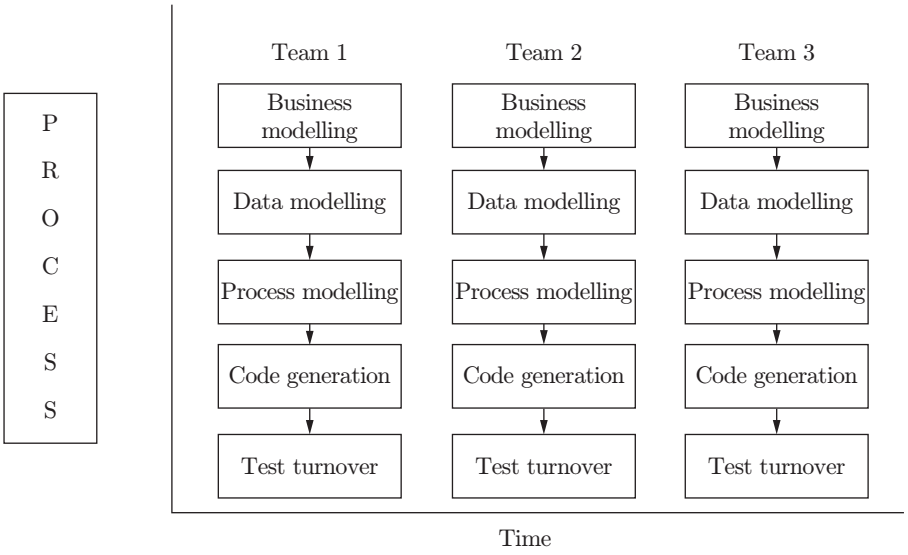


Figure 9.8 | RAD model.

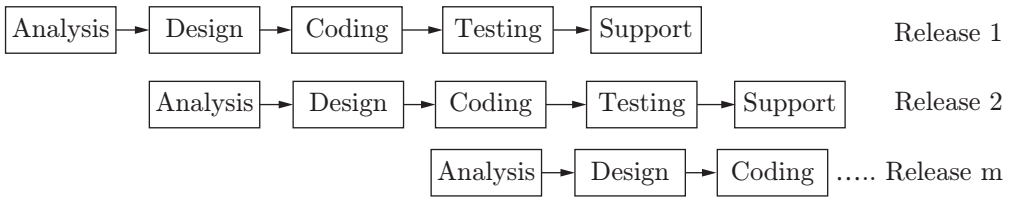


Figure 9.9 | Incremental model.

9.4.2.2 Spiral Model

Steps involved in spiral model (Fig. 9.10) are as follows:

Step 1: Customer communication

Step 2: Planning

Step 3: Risk analysis

Step 4: Engineering

Step 5: Construction and release

Step 6: Customer evaluation

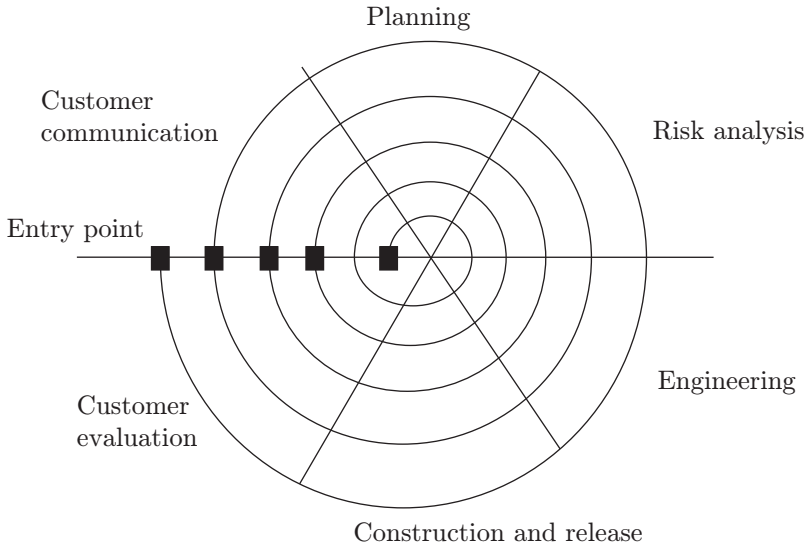


Figure 9.10 | Spiral model.

1. In the spiral model, the entry point is the customer communication to finalise the SRS. In the planning stage, we can prepare the estimation model to estimate the cost schedule and performance.
2. In the risk analysis stage, we can identify the technical and management risks. In the engineering stage, we can select the effective process model to develop the software.
3. In the construction and release stage, the procedural description is converted into machine-readable format and test cases also implemented. Later, the software is delivered to the customer.
4. In the customer evaluation stage, the software undergoes operation. During operation the customer identifies the change requirement and posts it to the development team. The change is reflected in the next release of the software.
5. In the spiral model, customer satisfaction is high, and at the same time the designer is able to design efficient design templates and deliver the code on budget. So, this model is called win-win spiral model.

9.4.2.3 Component-Based Development Model

In the component-based model, the components are used in the project development (Fig. 9.11).

After designing the project, before constructing the final software check the availability of the components in the library. If the component is available, use it in the current project; if not, construct the new component and place it in the library, then move to the next iteration.

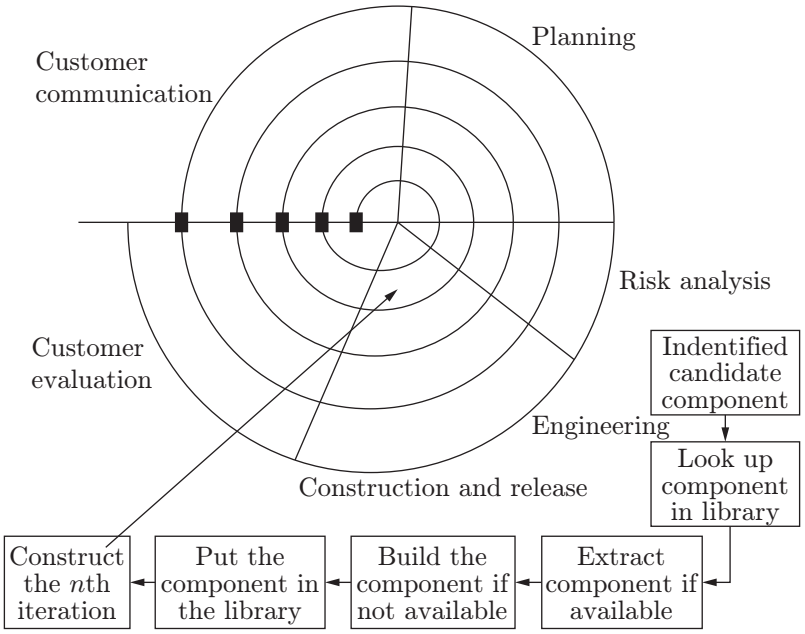


Figure 9.11 | Component-based development (CBD) model.

9.5 MEASUREMENT OF METRICS

Measurement is a quantitative indication which shows the size or complexity of a process.

9.5.1 Metrics

Metrics is a quantitative measure of an attribute of the project or process.

Measurement is divided into two types:

1. **Direct measurement:** In the direct measurement, the value is calculated based on direct approach. Example: Size of a project can be calculated based on the LOC (line of code).
2. **Indirect measurement:** Indirect measurement means the value can be calculated based on the other parameters. Example: Size of a project can be calculated by using the function point.

Software supports with four kind of P's:

1. People
2. Product
3. Project
4. Process

On the basis of the four P's, the software metrics are classified into the following three types:

1. **Product metrics:** This metrics describes the characteristics of the product such as size, complexity, performance and quality level.
2. **Project metrics:** It describes the project characteristics and execution. For example, effort cost and schedule; effort in terms of man-months. Effort of a project is 3 man-months means one developer works 3 months to develop the software.
3. **Process metrics:** It describes the characteristics of the process model to develop and maintain the software (Fig. 9.12). For example, calculates the defect rate and defect removal efficiency.

All the estimation models are empirical modules, that means, the models are developed based on the past experience without proof. Even though these models lead to success scenario of the software development.

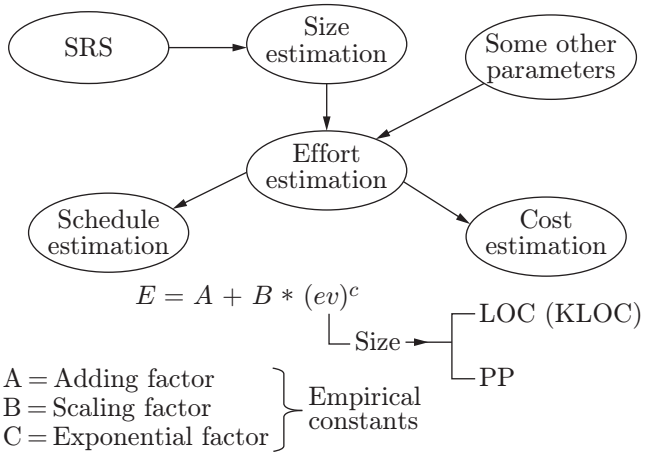


Figure 9.12 | Process metrics — an illustration.

9.5.2 Size-Oriented Metrics

The size of a project can be calculated using the following two ways:

1. Line of code (LOC)
2. Function point analysis (FPA)

9.5.2.1 Line of Code

If we want to estimate the size of a project, there is a need of taking the past project experience. Assume the LOC into three categories:

1. Optimistic LOC (S_{opt})
2. Most likely LOC (S_m)
3. Permissible LOC (S_{pess})

After identifying three levels of the source code, we can estimate the size of the project by using this following formula:

$$S = \frac{S_{opt} + 4S_m + S_{pess}}{6}$$

Problem 9.1: Consider a three-dimensional (3D) geometric application. The range of LOC estimated is $S_{opt} = 5600$, $S_m = 7900$ and $S_{pess} = 9600$. What is the expected size?

Solution:

$$S = \frac{5600 + 31600 + 9600}{6} = \frac{46800}{6} = 7800$$

So, the estimated size = 7800 LOC = 7.8 KLOC

9.5.2.2 Function Point Analysis

Function point analysis (FPA) is an indirect measurement to calculate the size of the project. Therefore, there is a need to consider the other parameters.

The function points are calculated based on the following five attributes:

1. Number of user input
2. Number of user output
3. Number of user enquiries
4. Number of files
5. Number of external interface

Every input is associated with the weighing factor. The weighing factor is defined based on the experience of the past project.

If the project is simple, the corresponding values are maintained in a table (Table 9.1). If project is complex, those values are also maintained in table.

Table 9.1 | Function point estimation

Attribute	Value	Weighing Factor		
		Simple	Average	Complex
I/P files		3	4	5
O/P files		4	5	7
Enquiries		3	4	6
Files		7	10	15
External interfaces		5	7	10

Count value = Values × Weighing factor

FP = Count value × VAF

Apart from the functional requirements, some other parameters are also involved in the project. So, we need to consider the effects of these factors on the project.

$$\text{VAF} = 0.65 + 0.01 \times \sum_{P=1 \text{ to } 14} F_i$$

where VAF is value adjustment factor.

Note: On the basis of the knowledge of the previous project, the weighing table is maintained as a simple/average/complex case. Calculate the count value by providing the multiplication between attribute values with the corresponding complexity levels. Default value of $\sum_{P=1 \text{ to } 14} F_i = 42$, if value adjustment complexity values are not given, use the default value.

Problem 9.2: Consider the software project with the number of input and output, enquiries, files and interfaces as 30(4), 25(5), 20(4), 10(10), 5(7). Consider () is a weighing factor. Assume all complexity adjustment factor are average = 3. Calculate the FP for the project.

Solution:

Values	Weighing Factor	Total
40	4	160
20	5	100
25	4	100
15	10	150
5	7	35
		Grand total = 545

Value of weighing factor = 545

FP = 545 × [(0.65) + 0.01 × 42] = 583.15

Problem 9.3: Consider 3D geometry application. The size of the project can be measured in terms of FP with the following parameters:

USER	
I/O	$3 \times 4 = 12$
O/P	$2 \times 5 = 10$
Enquiries	$2 \times 4 = 8$
Files	$1 \times 10 = 10$
External interfaces	$4 \times 7 = 28$

Assume the project complexity is average and value adjustment complexity is 1. Calculate the FP.

Solution:

$$\begin{aligned}\text{FP} &= 68 \times (0.65 + 0.01 \times 14)\\ &= 68 \times (0.65 + 0.14)\\ &= 68 \times (0.79)\\ &= 53.73\end{aligned}$$

Problem 9.4: The effort of the above project is 5 person-months. What is the productivity?

Solution:

Given that 5 persons-month $\Rightarrow$ 53.72

5 person-months means that the work done by 5 persons in 1 month.

Person 5 $\rightarrow$ 53.72

Person 1 $\rightarrow$ 53.72/5 = 10.74

Productivity = 10.74

or we can say, 10.74 functional point developed by 1 person-month.

9.5.3 Effort and Schedule (Duration) Estimation

Effort and duration can be estimated by using different empirical model structure.

1. According to the software engineering laboratory (SEL) empirical model, the effort can be estimated as follows:

   $$E = 1.4\ (\text{KLOC})^{0.93} \text{ person-months (units)}$$

   The duration can be calculated as

   $$D = 4.6\ (\text{KLOC})^{0.26} \text{ months (units)}$$

2. According to Walstonald-Felio method (W-F) (IBM):

   The effort can be calculated as

   $$E = 5.2\ (\text{KLOC})^{0.91} \text{ person-months}$$

   Duration can be calculated as

   $$D = 4.1\ (\text{KLOC})^{0.36}$$

Problem 9.5: Software development is expected to involve eight parameters of efforts. Calculate the following using SEL and WF models.

(a) The number of lines of code that can be produced.
(b) The duration of the development
(c) The productivity in LOC per person-year
(d) The average manning (person)

Solution:

By SEL method:

(a)
$$E = 1.4\ (\text{KLOC})^{0.93}$$
$$8P - 4 = 1.4\ (\text{KLOC})^{0.93}$$
$$(\text{KLOC})^{0.93} = \frac{(8 \times 12)}{1.4} = \frac{96}{1.4}$$
$$\text{KLOC} = \left(\frac{96}{1.4}\right)^{1/0.93} = 90.0414$$

So, LOC = 90041

(b) $D = 4.1\ (\text{KLOC})^{0.36} = 4.6 \times (90.0414)^{0.26}$
$= 4.6 \times 3.22 = 14.8 \sim 15$ months

(c) Productivity $= \dfrac{90041}{8} = 11255$

(d) Manning = Effort/Duration = $(8 \times 12)/15$ person-months = 6.4 persons

By WF method:

(a)
$$E = 5.2\ (\text{KLOC})^{0.91}$$
$$8P - 4 = 5.2\ (\text{KLOC})^{0.91}$$
$$(\text{KLOC})^{0.91} = \frac{(8 \times 12)}{5.2} = \frac{96}{5.2} = 18.46$$
$$\text{KLOC} = \left(\frac{96}{5.2}\right)^{1/0.91} = 24.63$$

So, LOC = 24630

(b) $D = 4.6\ (\text{KLOC})^{0.26} = 4.6 \times (24.63)^{0.26}$
$= 4.6 \times 2.3 = 10.6 \sim 11$ months

(c) Productivity $= \dfrac{24630}{8} = 3078.75$

(d) Manning = Effort/Duration = $(8 \times 12)/11$ person-months = 8.7 persons

9.5.3.1 Constructive Cost Model

By using constructive cost model (COCOMO), we can calculate the effort and time, this model is also an empirically derived model. Therefore, the structure of the effort calculation is

$$\text{Effort} = c_1 \times \text{EAF} \times (\text{size})^{c_2} \text{ (person-months)}$$

$$\text{Duration} = c_3 \times (\text{Effort})^{c_4} \text{ (months)}$$

where c_1 and c_3 are the empirically derived scaling factors and c_2 and c_4 are empirically derived exponential

factors. Size indicates in terms of KLOC or functional point. EAF (effort adjustment factor) (1 to 14 factors) rating (1–7) range.

COCOMO model is applicable in three different kinds of applications, as follows:

1. Organic mode (low complexity)
2. Semi-detached mode (average complexity)
3. Embedded mode (high complexity)

On the basis of the type of project, the constants are derived as follows:

	c_1	c_2	c_3	c_4
Simple	2.4	1.05	2.5	0.38
Average	3.0	1.12	2.5	0.35
Complex	3.6	1.20	2.5	0.32

Problem 9.6: A company needs to develop digital signal processing software for one of its newest invention. The software is expected to have 40000 LOC. The company needs to determine the efforts in person-months needed to develop this software using the basic COCOMO model.

The multiplicative factor for this model is given as 2.8 for the software development on the embedded system while exponential factor is given as 1.20. What is the estimated effort in person-months?

(a) 234.35 (b) 932.50
(c) 287.8 (d) 122.4

Solution:

Effort = 2.8 EAF $\times$ $(40000)^{1.20}$(EAF = by default =1)

$= 2.8 \times (40000)^{1.20}$

$= 2.8 \times (40)^{1.20}$

$= 234.35$

Problem 9.7: A company develops software for a digital signal processing application. The software size is expected to have 100000 LOC. LOC is sometimes called as DSE (delivery of source instructions).

The company needs to determine the effort in person-months and time in months to develop the software using COCOMO model under embedded mode. The effort adjustment factor is calculated based on the given table. What is the estimated effort and time for the above project.

S.No.	Cost Driven	Effect
1.	Language experience	1.0
2.	Schedule constraints	1.0
3.	Database size	1.0
4.	Turnaround time	1.0
5.	Virtual machine exp.	1.0
6.	Virtual machine volatility	1.0
7.	Use of software tools	0.88
8.	Modern programming practices	1.0
9.	Storage constraints	1.0
10.	Application experience	1.10
11.	Timing constraints	1.0
12.	Requirement reliability	1.15
13.	Product complexity	1.15
14.	Team capability	1.0

Solution:

AF = Multiplication of all cost-driven effect

(Multiple all) EAF = 1.28

$$E = c_1 \times \text{EAF} \times (\text{size})^{c_2}$$

As c_1= 2.0, so c_2 = 1.2 = 2.8 $\times$ 1.28 $(100)^{1.2}$ = 900 (person-months)

$D = 2.5 \times (\text{Effort})^{0.32} = 2.5 \times (900)^{.32} = 22.04$ months $\sim$ 23 months

9.5.3.2 Defect Rate

It is a quality metric, defect is undiscovered error during the development of the software. The defect rate is defined as

$$\text{Defect rate} = \frac{\text{Number of defect} \times \text{Time}}{\text{Total or number of LOC}}$$

9.5.3.3 Defect Removal Efficiency

Defect removal efficiency (DRE) is also a quality metric. It is defined as

$$\text{DRE\%} = \frac{E}{E + D} \times 100$$

where E = error and d = defect.

9.5.3.4 Halstead Size-Oriented Metric

Halstead metric is used to count the number of lines preset in the software. To calculate the size of the project in terms of LOC, we are supposed to count the number of lines in the software. But this value does not give the correct estimation because comments and spaces and non-executable lines are also counted.

LOC estimation means only the count of executable statements. To identify the correct executable statement in the software, there is a need to divide the program in the form of lexemes.

Lexemes	Token
For	Keyword
=	Equal -op
+	Add -op
*	Mul -op
3	Constant

After dividing the program into token, we can separate the operator and operands. Operators are special words and special symbol operators, etc. Operands are constant, variable, identifier, etc.

Unique Operator	Occurrence of the Operator in the Software	Unique Operand	Occurrence of the Operand in the Software
$\Sigma = n_1$	$\Sigma = N_1$	$\Sigma = n_2$	$\Sigma = N_2$

where n_1 = the count of unique operator used in the software

n_2 = count of unique operands used in the software

N_1 = count of occurrence of the operator in the software

N_2 = count of occurrence of the operands in the software

The following metrics are defined based on the above database.

The vocabulary can be calculated by using $n = n_1 + n_2$

The size of the software can be calculated by

$$N = N_1 + N_2$$

When we are considering machine language then

$$N = 2 \times \text{LOC}$$

The volume of the program can be calculated by using

$$V = N \log_2 n$$

The estimated level of program is defined as

$$L = \frac{2n^2}{N_1 \times N_2}$$

The difficulty is defined as

$$D = \frac{1}{L}$$

Error is defined as

$$E = \frac{V}{L}$$

Estimated program length is calculated as

$$n_1 \times \log_2 n_1 + n_2 \times \log_2 n_2$$

Problem 9.8: Consider the following program and calculate the software matrix n, N, V, D, E

```
var a, b, c, d, m: integer'
Begin
    Read in (a, b, c, d)
    a: = a + a;
    b:= b + b;
    c: = c * d;
    m: = a + b - c;
    write in (m);
End
```

Solution:

Unique Operator	Occurrence of the Operator in the Software	Unique Operator	Occurrence of the Operand in the Software
=	4	a	6
+	3	b	6
*	1	c	5
–	1	d	3
,	7		
:	1		
;	7		
( )	2		
.	1		
Read in	1		
Write in	1		
Integer begin-end	1		
n_1 =13	$N_1 = 31$	$n_2 = 5$	$N_2 = 23$

$$n = n_1 + n_2 = 13 + 5 = 18$$
$$N = N_1 + N_2 = 31 + 23 = 54$$
$$D = \frac{1}{L}$$
$$L = \frac{2n_2}{n_1 \times N_2} = \frac{2 \times 5}{13 \times 23} = \frac{1}{0.033} = 30.30$$
$$V = n \cdot \log_2 n = 54 \cdot \log_2 18 = 54 \times 5 = 270$$
$$E = \frac{V}{L} = \frac{270}{0.033} = 8181.81$$

9.6 RISK ANALYSIS

Risk means extreme condition when the product becomes a failure. Risk is associated with two characteristics:

1. **Uncertainty:** It means may/may not occur, 100% probable risk.
2. **Loss:** It means when the risk becomes real there is a loss, otherwise no loss.

9.6.1 Risk Identification

After identifying the extreme condition, there is a need of analyses of the risk in terms of level of uncertainty and degree of loss. The level of uncertainty is maintained in the following four ways:

1. Negligible
2. Marginal
3. Critical
4. Catastrophic

We are going to identify critical and catastrophic risk condition and try to manage them in the software execution with the help of risk strategy.

9.6.1.1 Risk Strategy

There are two types of risk strategies:

1. **Reactive risk strategy:** It means monitoring the project for the probable risk. When the risk is present during execution of the project, it implements the plans to control the risks. It is also called as fire-lighting mode.
2. **Proactive risk strategy:** It means risk plans and actions are developed before the implementation of the software. So, by using these strategies we can avoid the risk, if possible, or manage the risk, if not possible. By combining both proactive and reactive strategies, the risk mitigation, monitoring and management (RMMM) plan is prepared.

9.6.1.2 Types of Risk

1. **Project risk:** This risk threatens the project plan, therefore schedule will be extended and development cost will increase.
2. **Technical risk:** These risks threaten the quality and timeliness of developing the software. When this risk becomes real, we cannot develop the project within the quality levels.
3. **Business risk:** These risks threaten the viability of software risk. Under this category, four different risks are defined:
   - *Market risk:* Build an excellent product but no users.
   - *Strategic risk:* Build a product that is no longer fit into the business objective.
   - *Management risk:* Lack of high-level support to inject the changes.
   - *Budget risk:* No synchronisation between cost and price.
4. **Known risk:** Uncover errors after a careful evaluation of the project.
5. **Predictable risk:** It is extrapolation from the past project experience.
6. **Unpredictable risk:** It acts as a joker in the deck.
7. **Generic risk:** It describes the characteristics of the project such as planning, performance, cost, effort, etc.
8. **Product-specific risk:** These risks describe the characteristic of the product such as the size, quality, performance, etc.

Risk Components

There are four different components present in the software to maintain the probability of probable risk:

1. **Performance:** It indicates the level of uncertainty to meet its specification.
2. **Cost:** It indicates the level of uncertainty to maintain project budget.
3. **Support:** It indicates the level of uncertainty to easily correct, adopt or enhance the existing software.
4. **Schedule:** It indicates the level of uncertainty to maintain the schedule and quality of the software.

9.6.2 Risk Projection or Risk Estimation

To estimate the cost of the risk, there is a need to identify the probability of the risk becoming real and the associated impact.

Risk Name	Category	Probability	Impact

By using the above database we can estimate the risk cost by using the following formula:

$$\text{Risk exposure (RE)} = P \times C$$

where P is the probability of risk becoming real and C is the cost.

Problem 9.9: To develop an image-processing application, the software uses component-based development approach '60'. Reusable software component are planned but 70% of the project reusable component are used. '18' components are developed from the baseline to complete the project. The cost component size is '100' LOC, and the cost for each LOC is \$14. What is the risk RE when the risk probability is 80%?

Solution:

$$\text{RE} = P \times C$$
$$P = 80\%$$
$$C = 18 \times 100 \times 14 = \$252000$$
$$\text{RE} = 0.8 \times 25200$$
$$= \$20160$$

9.7 SOFTWARE DEVELOPMENT LIFE CYCLE

The software development life cycle (SDLC) describes the generic approach to develop the software. The framework of the SDLC is shown in Fig. 9.13.

9.7.1 Requirement

According to the IEEE standard, the requirement is defined as a condition or capability needed by a user to solve a problem or to achieve an objective.

There are five types of requirements classified in the software engineering (Fig. 9.14):

1. **Business requirement:** These requirements describe the high-level business requirements. These requirements are documented in vision and scope document.

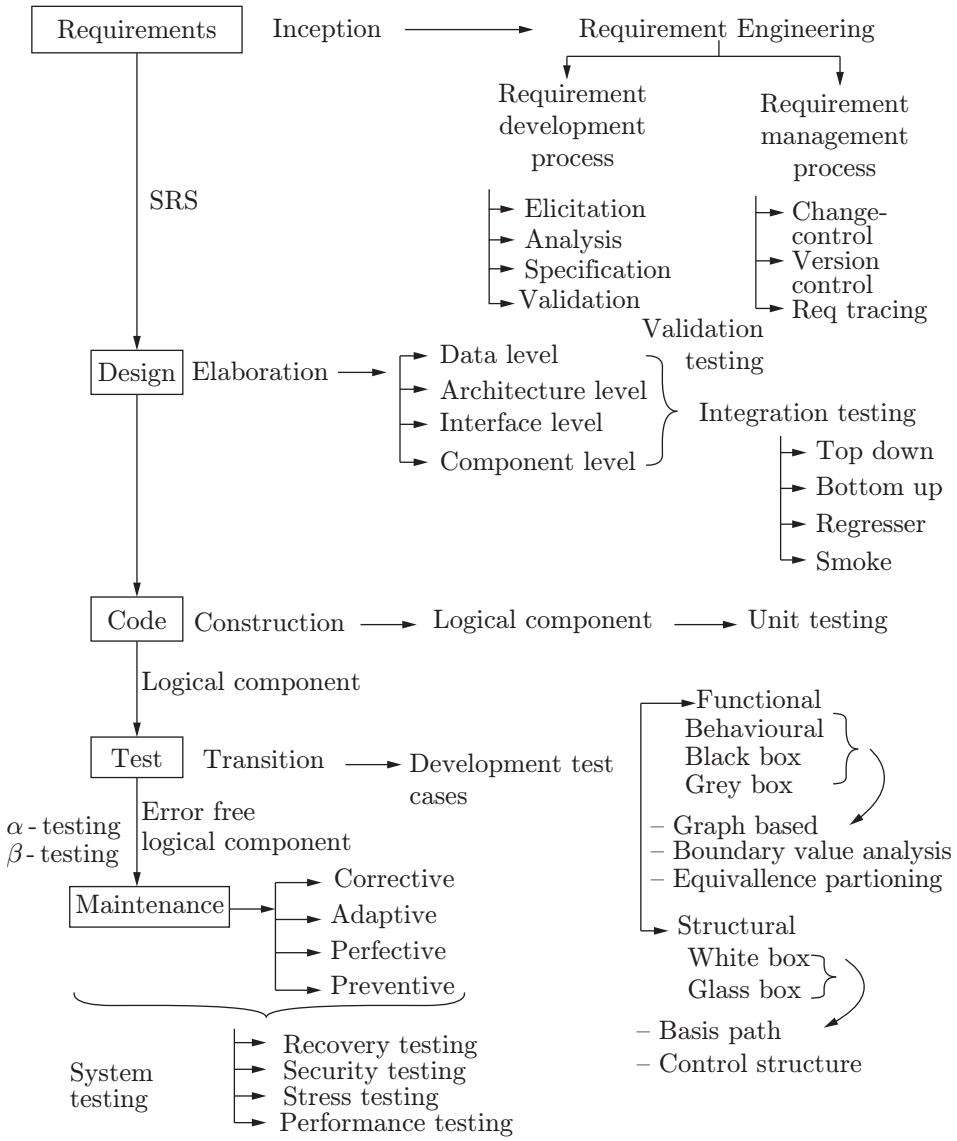


Figure 9.13 The software development life cycle.

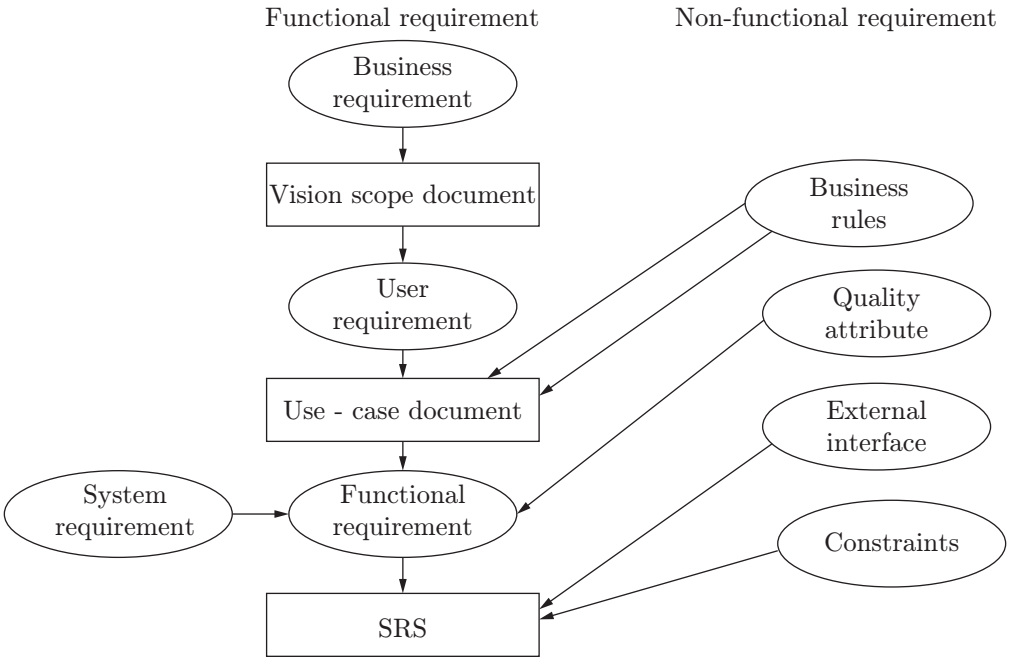


Figure 9.14 | Requirement phase of SDLC.

2. **User requirement:** This requirement describes the user need, user goal and user capability towards the system to achieve their objective. Those requirements are documented in use-case document.
3. **Functional requirements:** This requirement describes the functionality of the software, which is developed into the system to satisfy the user requirement. These requirements are documented in the SRS document.
4. **System requirement:** This requirement describes the subsystem requirement on which platform the developed product is running, that is, CPU, memory capacity and operating system. These are the subset of the functional requirement.
5. **Non-functional requirement:** This requirement describes the business rules (such as govt. acts, govt. policies, account plans) and quality attributes (availability, usability, maintainability, robustness, flexibility).

9.7.1.1 External Interfaces

Like how many other systems are communicating with the software and constraints (risk condition). All these requirements are documented in the SRS.

Note: SRS document consists of goals of the software, functional requirement and non-functional requirements.

In order to develop the right system, the right requirements need to be injected into the software development process, and for developing the right requirement there is a need of requirement engineering.

9.7.1.2 Requirement Engineering

Requirement engineering includes the systematic, disciplined, quantifiable approach to the development, operation and maintenance of the requirement. The following two processes take place in requirement engineering:

1. **Requirement management process:** These activities are used to manage the requirement throughout the project development and lifetime of the product.
2. **Requirement development process:** The framework used to develop the requirement is shown in Fig. 9.15.

On the basis of the goal of the software, elicit the requirements from different sources and create the data dictionary.

On the basis of the data dictionary, create the structure of the software. On the basis of the structure of the software, prepare the technical specifications. All the customer capabilities are maintained in the SRS document.

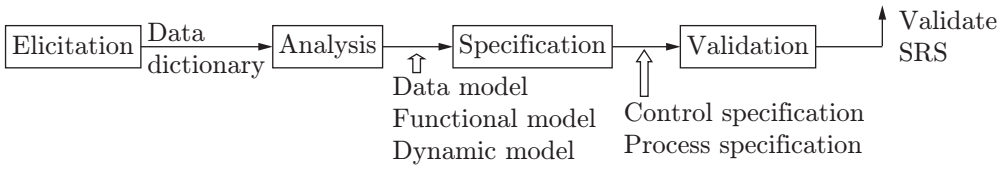


Figure 9.15 | Requirement process.

After reviewing the SRS document twice or thrice, the customer and the developer sign on the SRS document. This document (SRS) is then forwarded to the design face.

9.7.1.3 Elicitation

In this state, by using different techniques, elicit the requirement from different sources towards goal of the project. The different elicitation techniques are as follows:

1. **CORE (Controlled Requirement Expression):** In CORE technique, elicit the requirement by using view point analysis. In this analysis, maintain the sequence of data object, action and consequence.
2. **IBIS (Issue-based Information System):** In IBIS technique, elicit the requirement based on the question and answer analysis. This technique maintains the issue, position and justification.
3. **FODA (Future-oriented Domain Analysis):** In FODA technique, elicit the requirement by creating different models to specify the structure of the end product. This technique maintains the sequence context model — function model — dynamic model (behaviour or data model).
4. **QFD (Quality Function Deployment):** In QFD technique, elicit the requirement towards quality deployment.
5. **JAD (Joint Application Development):** In JAD technique, elicit the requirement by conducting a quick meeting between customer, end user, analyst and developers.
6. **Prototype:** In prototype technique, elicit the requirement by creating the quick design mock-ups and taking continuous feedback from the customer.

Note: The output of the elicitation stage is data dictionary, it consists of data objects related to the software.

9.7.1.4 Analysis

This stage describes the structure of the end product in terms of data object, functionality, information flow and behaviour with respect to external event, by creating three types of model:

1. Context or data model (E-R diagram)
2. Functionality and information flow model (described by DFD)
3. Dynamic or behavioural model (described by STD)

1. **Context or data model (E-R Diagram):** This model describes the structure of the end product in terms of data object and relationship. Data model can be represented by using the entity, relationship diagram. E-R diagram consists of object, attribute and relationship.

   Object is defined as a representation of composite elements. Composite elements must have multiple properties. The property of the object is called as attribute. The relationship can be maintained between the object by using two concepts.

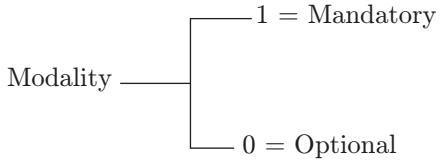
   - **Cardinality:** It indicates in how many ways one object can be communicated with another object. There are three types of cardinality:

     1: 1

     1: n

     M: N

   - **Modelling:** It indicates "is the communication link between the object mandatory or optional".



   Note: On the basis of the data model, the data structure is created in the data level design.

2. **Functional and information flow model:** The functional and information flow model describes the functionality of the end product, that means, what is the behaviour of the end product with a proper set of input. The functionality can be represented by using the data flow diagram (DFD). Data flow diagram is also called as Bubble chart or directed graph. Different levels of abstraction of the end product can be represented using different levels of DFD.

   - **Level-Zero DFD:** It describes the high-level abstraction of the functionality by maintaining the single process bubble with multiple input- and output-directed edges (Fig. 9.16).

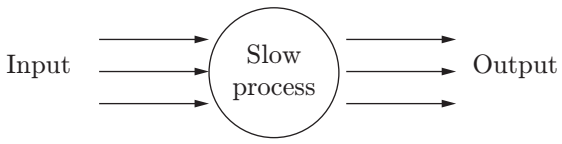


Figure 9.16 | Level-zero DFD.

   In this level of DFD, only the functionality is represented with absence of information flow.

   - **Level-1 DFD:** It describes the functionality and information flow of the software by maintaining different process bubbles. It describes the project in low-level abstraction. Low-level abstraction is easy to code (Fig. 9.17).

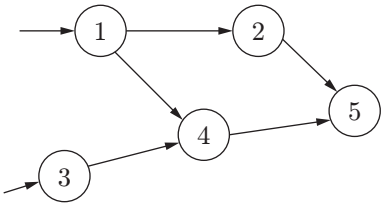


Figure 9.17 | Level-1 DFD.

3. **Dynamic model:** It describes the behaviour of the system with respect to external events. This model is represented using the state transition diagram (Fig. 9.18).

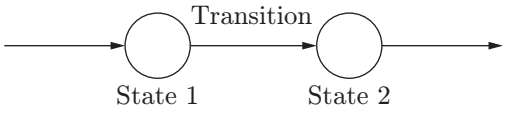


Figure 9.18 | State transition diagram.

- **Specification:** In this stage, the models are refined into control specification and process specification
  - **(a) Control specification:** It shows the behaviour of the product.
  - **(b) Process specification:** It refines the mathematical expression, algorithm and constraints.
- **Validation:** In this stage, various validation criteria are implemented to validate the requirement towards the right condition. The following reviews are done to validate the software:
  (a) Configuration review
  (b) Quick review
  (c) Detailed review

  "Validation means the black-box testing is performed at the requirement stage itself".
  After validating the SRS document, it is forwarded to the design stage. The output of the requirement development process is shown in Fig. 9.19.

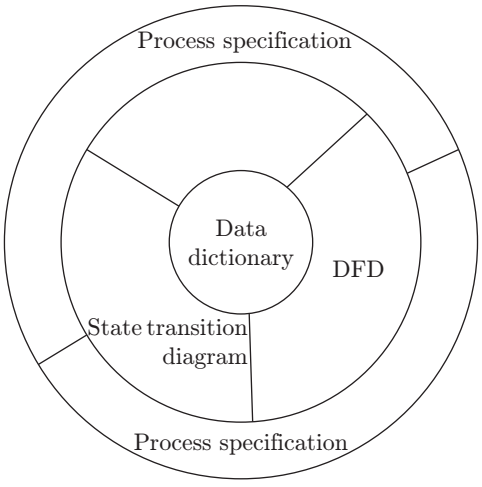


Figure 9.19 | Validation.

9.7.2 Design

The design stage represents the project in the low-level abstraction, that means, the customer description is translated into technical description. To do this process, the following four levels of design take place (Fig. 9.20):

1. **Data level:** Data level design describes the data structure. It takes help from the E-R diagram and data dictionary.
2. **Architecture level:** It defines the structure of the end product (skeleton). It takes help from the DFD of the last stage.
3. **Interface level:** Interface level design describes the number of interfaces required in the system. It takes help from the state transition diagram.
4. **Component level:** It represents the procedural description to implement the functionality in the software. It can take help from process specification and control specification (Fig. 9.21).

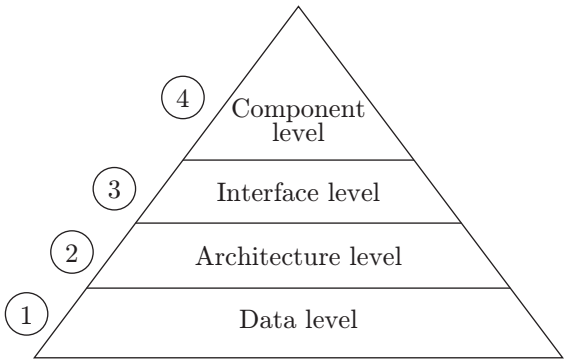


Figure 9.20 | Levels.

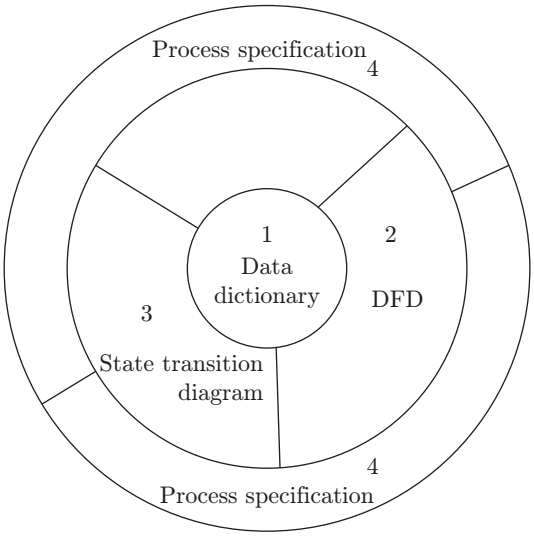


Figure 9.21 | Component.

9.7.2.1 Designing Concept

The following concepts should be kept in mind while designing a document:

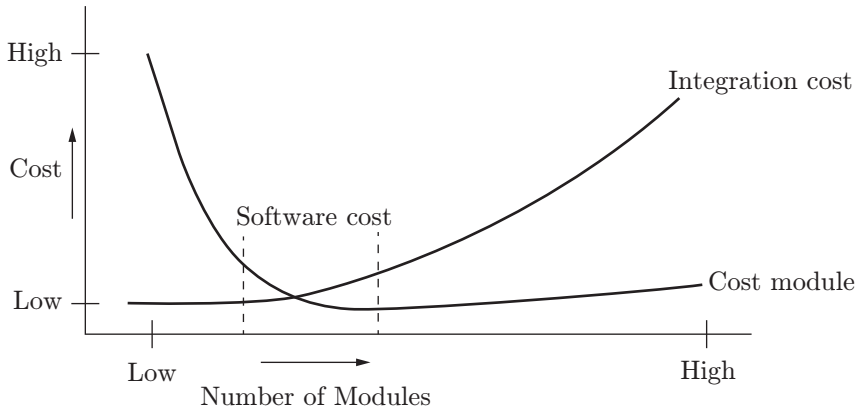


Figure 9.22 Number of modules versus cost of software development.

1. **Abstraction:** There are different levels of abstraction available to represent the statement. In high-level abstraction, the statement is represented in concrete manner, that means, internal structure is not provided. In low-level abstraction, all statements are defined in a single format with all details of implementation.
2. **Refinement:** It is a top-down strategy to simplify the statement structure. Refinement is also called as elaboration. After refinement of the statement, we can get the direct implementation statement.
3. **Modularity:** Software development process uses the modularity concept to reduce the development cost and time. Suppose the complexity of the problem is denoted by $C(P)$ and the effort of the problem is denoted by $E(P)$. The following formats represent the advantage of C.

   $C(P_1) \quad E(P_1)$
   $C(P_2) \quad E(P_2)$
   $C(P_1) > C(P_2)$
   $C(P_1) > E(P_2)$
   $C(P_1 + P_2) > C(P_1) + C(P_2)$
   $E(P_1 + P_2) > E(P_1) + E(P_2)$

   When the number of modules is less, the cost module for developing the module is high, and at the same time the integration cost is low.

   When the number of modules increases, the cost per module is low but integration cost is high. This relationship is shown in Fig. 9.22.

Modules should be designed in such a way that each module has specific functional requirements. Functional independence is measured using two terms cohesion and coupling.

9.7.2.2 Cohesion

It is a measure of relative functional strength of a module. Following types of cohesion exist:

1. **Logical cohesion:** Logical cohesion exists when a module that performs tasks are related logically.
2. **Temporal cohesion:** Temporal cohesion exists when a module contains tasks that are related in such a way that all the tasks must be executed within the same time limit.
3. **Procedure cohesion:** The procedural cohesion exists when the processing elements of a module are related and must be executed in a specific order.
4. **Communication cohesion:** The communication cohesion exists when all the processing elements concentrate on one area of a data structure.

9.7.2.3 Coupling

Coupling is a measure of relative independence among modules, that is, it is a measure of interconnection among modules. Following types of coupling exist:

1. **Data coupling:** Data coupling exists when a module accesses another module through an argument or through a variable.
2. **Stamp coupling:** Stamp coupling exists when a module accesses another module through a data structure.
3. **Control coupling:** Control coupling exists when the control is passed between modules using a control variable.
4. **External coupling:** External coupling exists when modules are connected to an environment external to the software.
5. **Common coupling:** Common coupling exists when the modules share the same global variable.
6. **Content coupling:** Content coupling exists when one module makes use of data or control information maintained in another module. Also, content coupling occurs when branches are made into the middle of a module.

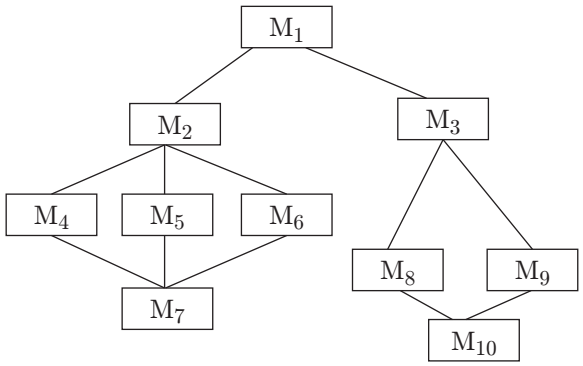
Note: Effective modular design must have high cohesion and low coupling.

9.7.2.4 Software Architecture

It shows the hierarchical structure of the software and also describes the logical connections between the modules. In this process, it identifies the data model, functional model and behaviour model.

Control Hierarchy

It indicates the control sequence in the software project. It represents the interaction between different modules (Fig. 9.23).



Fan-in ⇒ number of control in — Fan-in $(M_7) = 3$

Fan-out ⇒ number of control out — $(M_2) = 1$

Fan-out $(M_7) = 0$

$(M_2) = 3$

Figure 9.23 An illustration of control hierarchy.

The control hierarchy defines various parameters of the software design:

1. **Depth:** It is a measure of the number of levels present in the software design.
2. **Width:** It is a measure of the maximum number of modules present at any one of the level.
3. **Fan-in:** It is a measure of the number of modules already controlling the given module.
4. **Fan-out:** It is a measure of the number of modules controlled by the given module.
5. **Structural partitioning:** It indicates the control sequence of development, maintenance, and integration of the software. It is divided into the following two types:
   - Horizontal
   - Vertical

Horizontal partitioning allows the level by level development and integration, that means, control flow is assigned to levels.

Vertical portioning means the control sequence is forwarded from root to leaf node, that is, depth-wise development and integration take place. Because of dependencies, this approach is not suggested to software development.

There are different advantages with horizontal partitioning:

1. Easy to develop
2. Easy to integrate
3. Easy to test
4. Easy to maintain

Data Structure

It shows the logical connection between the data objects.

Software Procedure

It describes the customer requirements into programming language statements. Information hiding is used in effective modularity. It shows the independence of the module. After the designing stage, we have the software design document (SDD). The SDD represents the programming description language oriented customer objective.

9.7.3 Coding

It translates the procedural-oriented description into machine-readable format by using appropriate programming language.

9.7.4 Testing

The objective of testing to measure to what extent the developer logical program becomes a failure. In the testing phase, different test cases are developed and implemented to cover the structural and functional errors. Functional testing includes conducting tests to check the functionality of the product with all proper sets of input, which also covers errors. Structural testing includes conducting tests to examine the internal structure of the developed product to cover the error.

9.7.4.1 Structural Testing

This testing is also called as white-box or glass-box testing. The objective of white-box testing is as follows:

1. Guarantee the all the statements present in the developed program exercise at least once.
2. Exercise logical expression towards true and false side.
3. Exercise all the loop boundaries within the operational limit and beyond the operational limit.
4. Exercise the data structure present in the program towards validity.

To reach the objectives of white-box testing, two different test operations are performed:

1. Basis path testing
2. Central structural testing

Basis Path Testing

This test is used to cover all the paths present in the development code. This testing guarantees that all the statements in the program must be exercised at least once to identify the path in the program, convert the program into graphical notation. On the basis of the control flow between the statements, the program is converted into graph notation called as flow graph or program graph. Flow graph is a directed graph.

Each node in the flow graph represents the program statement. The sequence of the statement is represented by using the following (Fig. 9.24):

Figure 9.24 | Sequence statement.

1. The if condition is represented by Fig. 9.25.

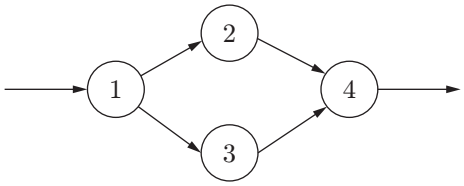


Figure 9.25 | IF condition.

2. The while statement is represented by Fig. 9.26.

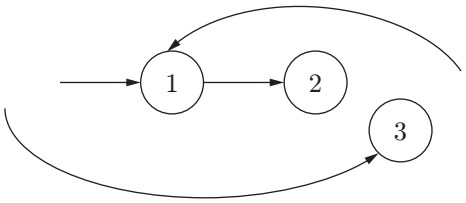


Figure 9.26 | WHILE statement.

3. Do–while statement is represented by using Fig. 9.27.

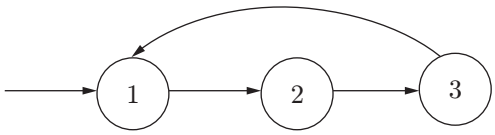


Figure 9.27 | DO–WHILE statement.

4. The switch case statement is represented by Fig. 9.28.

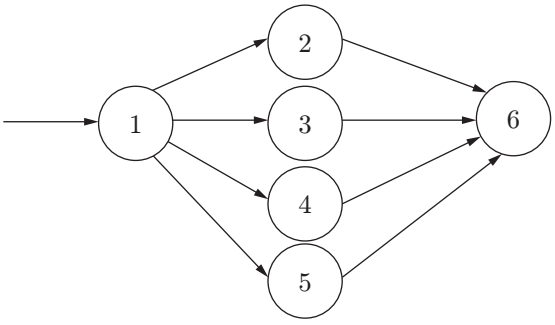


Figure 9.28 | SWITCH-CASE statement.

By using the cyclomatic complexity we can measure the logical complexity of the program.

Cyclomatic Complexity

It is defined as the quantitative measure of logical complexity of the program. Cyclomatic complexity can be calculated in three ways:

1. By using the number of edges and number of nodes we can measure the cyclomatic complexity as

   $V(G) = E - N + 2P$

   $E =$ Number of edges

   $N =$ Number of nodes

   $P =$ Number of disconnected component

2. By using the region we can estimate the cyclomatic complexity as

   $V(G) =$ Number of regions present in the graph

   When we count the cyclomatic complexity of a program by using basis path testing, outside of graph is considered as one region.

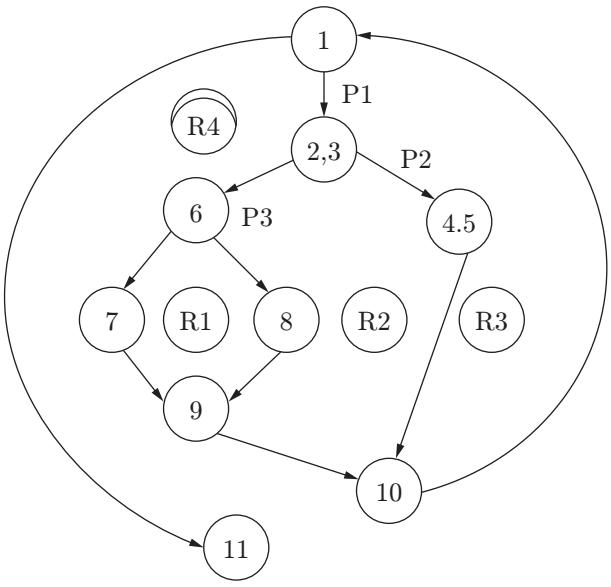
3. By using predictable nodes:

   $V(G) = P + 1$

   $P =$ Number of predicate nodes

   Predictable nodes are calculated by two or more nodes expanding from it.

Problem 9.10: Consider the following flow graph and calculate the cyclomatic complexity and also identify the independent path.



Flow graph depicting relation among different nodes.

Solution:

Cyclomatic complexity $= E - N + 2$

$= 11 - 9 + 2 = 4$

Number of regions $= 4 = R_1, R_2, R_3, R_4$

So, cyclomatic complexity $= 4$

Cyclomatic complexity $P + 1 = 3 + 1 = 4$

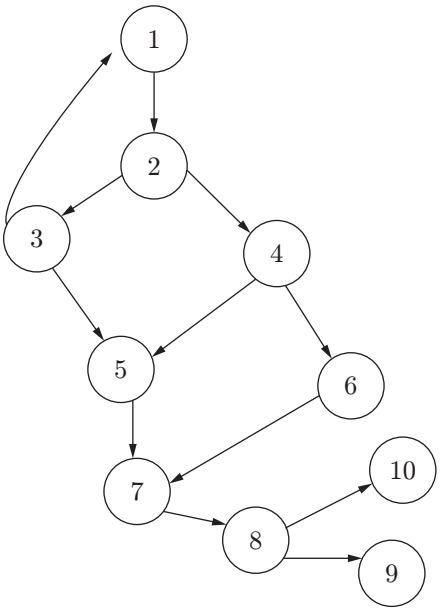
Path 1: 1-11

Path 2: 1-2-3-4-5-10-1-11

Path 3: 1-2-3-6-8-9-10-1-11

Path 4: 1-2-3-6-7-9-10-1-11

Problem 9.11: Consider the following program and calculate the cyclomatic complexity.



Flow graph depicting relation among nodes.

Solution:

Rule 3: $P + 1 = 4 + 1 = 5$

Rule 2: Number of regions $= 5$

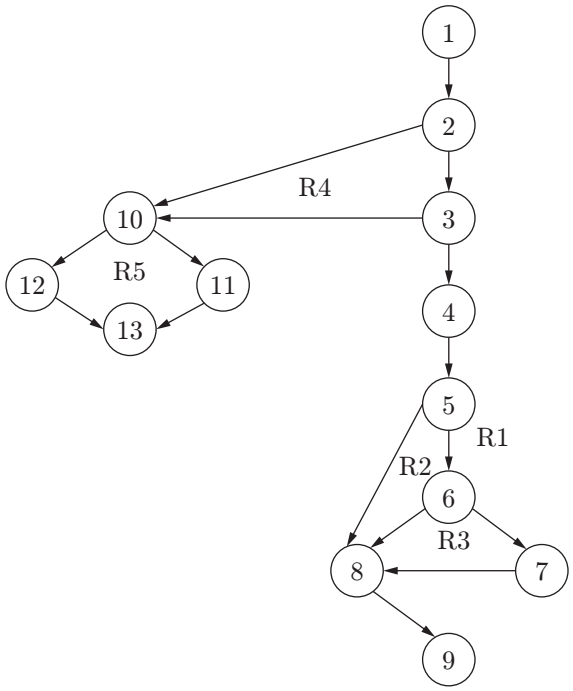
Rule 1 $= E - n + 2 = 13 - 10 + 2 = 5$

Path 1: 1-2-45-6-7-8-10

Path 2: 1-2-4-5-7-8-10

Path 3: 1-2-3-5-7-8-10 and more

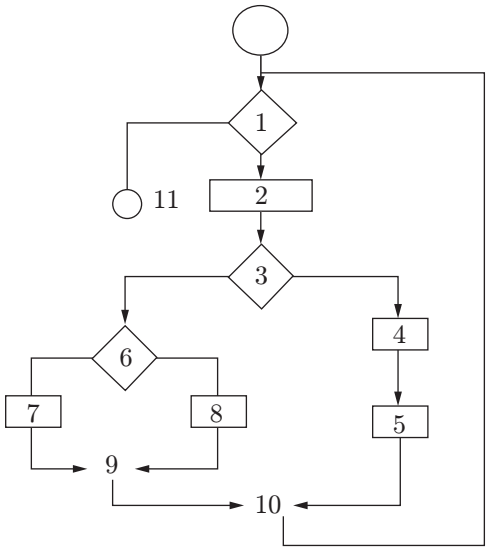
Problem 9.12: Consider the following graph and calculate the cyclomatic complexity.



Flow graph depicting relation among nodes.

Cyclomatic complexity $= 6$

Problem 9.13: Consider the following flow chart and calculate the cyclomatic complexity.



Flow chart depicting various operations of a program.

Cyclomatic complexity = Number of if statement + 1 = 3 + 1 = 4

The cyclomatic complexity of the program can be calculated by using the graph matrices.

Graph Matrices

The flow graph is converted into graph matrices by maintaining rows and columns. The number of rows and columns depends on the number of nodes in the flow graph. Based on the communication link between the nodes in the flow graph, the entry is reflected in corresponding position in the graph matrices.

If any row consists of more than one entry that node is called as predicate node, then take the weightage of predicate node by subtracting one from the total. After subtraction operation, add all the row values, it gives the predicate nodes. Use the formula to calculate cyclomatic complexity $V(G) = P + 1$.

Example 9.1

	1	2	3	4	5
1		b		c	1
2	a		d		
3		e		f	
4			g		
5	h				

Connection Matrix ⇒

	1	2	3	4	5
1	0	**1**	0	**1**	**1**
2	**1**	0	**1**	0	0
3	0	**1**	0	**1**	0
4	0	0	**1**	0	0
5	**1**	0	0	0	0

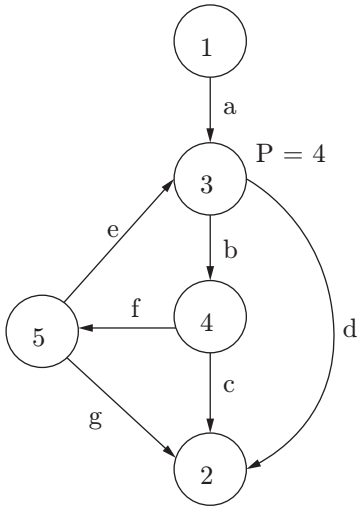
$3 - 1 = 2$
$2 - 1 = 1$
$2 - 1 = 1$
$1 - 1 = 0$
$1 - 1 = 0$
$P = 04$

	1	2	3	4	5
1		**1**		**1**	**1**
2	**1**		**1**		
3		**1**		**1**	
4			**1**		
5	**1**				

A graph matrix and its allocation.

Problem 9.14: Consider the following flow graph and create the graph matrix and calculate the cyclomatic complexity.

Note: If there are more than two outgoing nodes, then the predicate formula will not work.



Mode \ Link	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		h

A flow graph and graph matrix. Solution:

So, cyclomatic complexity = $P + 1 = 5$

Control Structure Testing

This testing is used to cover the error from the control structure present in the program. There are three types of expressions:

1. Arithmetic (= , − , *, /) = value
2. Relational (>, <, <=, >=, =, ≠) = True/False (Boolean)
3. Boolean (&&, ||, ¬, /) = True/False (Boolean data type)

Examples:

1. $a + b$
2. $(a + b) > (c + d)$ = T/F
3. $((a + b) > (c + d)$ && $((a \times b) < ((c * d))$ = T/F

After evaluating the control expression, the output may be a Boolean data, that is, true/false.

In the control structure testing, examine all the logical expression towards true/false. During that evaluation, it covers the Boolean operator error, Boolean data or parenthesis symbol error, relational operator error and arithmetic operator errors.

In this testing, two test cases are developed to cover all the errors described in the above list.

1. Branch testing
2. Domain testing

9.7.4.2 Black-Box Testing

In this testing, various input test cases are developed and the product functionality is examined, whether it satisfies all possible inputs. This testing is also called behavioural testing or grey-box testing. In this strategy, various testing techniques are used to cover the functional errors such as

1. Equivalence partition
2. Boundary value analysis
3. Comparison

1. **Equivalence partition:** In this testing, the input domain is divided into equivalent partitions. Each position is called as class. One partition is the exact valid range and another partition is above the upper limit and another is below the lower limit.

Example 9.2

If the input box consists of 1 to 1000 values, no need of preparing the thousand (1000) test cases. By using equivalence partition, the above input box is divided into three test cases.

Test case is prepared between the ranges of {1–1000}

Test case is prepared {>1000}

Test case is prepared {<1}

2. **Boundary value analysis:** Most of the problems are possible at the extreme edges that are boundaries of input domain. Boundary value analysis focuses on identified errors at the boundary rather than finding those existing at the centre of the input domain.

Example 9.3

Test case for input box accepting number (1—1000) using boundary value analysis is

(a) Test cases with data exactly the input boundary of input domain, that is, 1 and 1000.
(b) Test data with values at just below the extreme edges of input domain, that is, 0 and 999.
(c) Test data with values just above the input boundary of input domain, that is, 2 and 1000.

Note: When the product supports the N variable that is tested using the boundary value analysis, that is, $4N + 1$ test cases.

9.7.4.3 Comparison Testing

When the software is developed in the real-time application domain, there is a predefined value estimated based on the product behaviour. After developing the software, compare the behaviour of the new product with predefined values and remove the errors.

9.7.4.4 Life Cycle Testing

Various testing plans are implemented at each stage of the software development life cycle.

9.7.4.5 Validation Testing

This testing is performed at requirement stage by conducting the black-box test plans on a prototype model.

9.7.4.6 Integration Testing

This testing is performed at the design stage. Software development uses the modularity concept and develops the module independently. After developing the module, there is a need of integration to generate the final system.

In that regard, integration testing is required to check the functionality. Integration testing is not focused on internal details of the modules.

1. **Top-down integration:** This testing focuses on the function of the module integration either depthwise or breadthwise. Depth-wise integration is a vertical partitioning and widthwise integration is horizontal partitioning.
2. **Bottom-up integration:** In this testing, the integration starts from leaf node and forwards to the root node. In this testing, the stubs and drivers are maintained.

9.7.4.7 Regression Testing

When a new module is added, there is a possibility of introducing new functionality to the existing system. Due to the new functionality, there is a possibility of errors occurring in the existing system. To cover those errors, all test cases are re-conducted. This is called regression testing.

9.7.4.8 Smoke Testing

This testing is used in the wrap-shrink component development. Wrap-shrink component development means within the short period of time the software is implemented. To perform this we use modularity and by adding or deleting the existing module we can develop the final product. In this way, smoke testing is used to cover the error.

9.7.4.9 Unit Testing

This testing is performed at the coding stage. After developing the module, use the white-box test plan to cover the structural errors in the module.

9.7.4.10 Alpha Testing

After developing the product, this test is conducted by the customer at the developer side to cover the error. In this testing, the developer's presence is there.

9.7.4.11 Beta Testing

After deploying the product to the customer side, this test is conducted by the customer or end user at the customer place. In this testing, the developer's presence is not there.

9.7.4.12 System Testing

After developing the product, various system-level testing operations are performed to check the reliability of the product.

1. **Recovery testing:** This testing focuses on the recovery of data due to crashes or power failure.
2. **Security testing:** This testing focuses on unauthorised accessing of product.
3. **Stress testing:** This testing is focused on load balancing. It is also known as volume testing.
4. **Performance testing:** This testing is a user acceptance testing in terms of quality and reliability.

IMPORTANT FORMULAS

1. Size $S = \dfrac{S_{\text{opt}} + 4S_{\text{m}} + S_{\text{pess}}}{6}$
2. FP = Count value × VAF
   where $\text{VAF} = 0.65 + 0.01 \times \sum_{P=1 \text{ to } 14} F_i$
3. According to the SEL empirical model, the effort and duration can be estimated as
   $E = 1.4\ (\text{KLOC})^{0.93}$ person-month (units)
   $D = 4.6\ (\text{KLOC})^{0.26}$ months (units)
4. According to Walstonald-Felio method (W-F) (IBM):
   $E = 5.2\ (\text{KLOC})^{0.91}$person-months
   $D = 4.1\ (\text{KLOC})^{0.36}$
5. COCOMO model
   $\text{Effort} = c_1 \times \text{EAF} \times (\text{size})^{c_2}$ (person-months)
   $\text{Duration} = c_3 \times (\text{Effort})^{c_4}$ (months)
6. RE = $P \times C$
   where P is the probability of risk becoming real and C is the cost.
7. Cyclomatic complexity
   $V(G) = E - N + 2P$, where E = number of edges, N = number of nodes and P = number of disconnected components.
   $V(G)$ = Number of regions present in the graph
   $V(G) = P + 1$, where P = number of predicate nodes

SOLVED EXAMPLES

1. The most important feature of spiral model is

   (a) Requirement analysis
   (b) Risk management
   (c) Quality management
   (d) Configuration management

   Solution: Risk management helps in developing a cost effective project where risks are analyzed and risk strategies are decided upon.

   Ans. (b)

2. Cyclomatic complexity of a flow graph G with n vertices and e edges is

   (a) $V(G) = e + n - 2$ (b) $V(G) = e - n + 2$
   (c) $V(G) = e + n + 2$ (d) $V(G) = e - n - 2$

   Solution: Cyclomatic complexity is a software metric developed by Thomas J. McCabe. It is used to indicate the complexity of a program. It directly measures the number of linearly independent paths through a program's source code. It is also known as conditional complexity. It is defined as $V(G) = e - n + 2$.

   Ans. (b)

3. Testing of software with actual data and in actual environment is called

   (a) Alpha testing (b) Beta testing
   (c) Regression testing (d) None of the above

   Solution: Users or an independent test team at the developer's site performs alpha testing. Alpha testing

is often employed as a form of internal acceptance testing, before the software goes to beta testing.

Beta testing is performed after alpha testing for user acceptance testing.

Regression testing focuses on finding defects after a major code change has occurred.

Ans. (b)

4. Object Request Broker (ORB) is

I. A software program that runs on the client as well as on the application server.
II. A software program that runs on the client side only.
III. A software program that runs on the application server, where most of the components reside.

(a) I, II and III (b) I and II
(c) II and III (d) I only

Solution: ORB manages interaction between clients and servers.

Ans. (d)

5. Key process areas of CMM level 4 are also classified by a process which is

(a) CMM level 2 (b) CMM level 3
(c) CMM level 5 (d) All of the above

Solution: There are five maturity levels in CMM model—(1) Initial, (2) Managed, (3) Defined, (4) Quantitatively Managed and (5) Optimizing. The organization at level 2 includes level 1. So, level 4 is included in level 5.

Ans. (c)

6. Validation means

(a) Are we building the product right
(b) Are we building the right product
(c) Certification of fields
(d) None of the above

Solution: Verification means are we building the product right, whereas validation means whether we are building the right product.

Ans. (b)

7. Function points can be calculated by

(a) UFP × CAF (b) UFP × FAC
(c) UFP × Cost (d) UFP × Productivity

Solution: Function Point FP = UFP × CAF. UFP stands for Unadjusted Function Point, while CAF stands for Complexity Adjustment Factor.

Ans. (a)

8. Superficially the term "object-oriented", means that, we organise software as a

(a) collection of continuous objects that incorporates both data structure and behaviour.
(b) collection of discrete objects that incorporates both discrete structure and behaviour.
(c) collection of discrete objects that incorporates both data structure and behaviour.
(d) collection of objects that incorporates both discrete data structure and behaviour.

Solution: This is the definition of object-oriented according to software engineering.

Ans. (c)

9. Which one of the following is not a key process area in CMM level 5?

(a) Defect prevention
(b) Process change management
(c) Software product engineering
(d) Technology change management

Solution: The key process areas at level 5 covers the following issues—Defect Prevention, Technology Change Management and Process Change Management. Software Product Engineering has been addressed in level 3.

Ans. (c)

10. Which of the following system components is responsible for ensuring that the system is working to fulfil its objective?

(a) Input (b) Output
(c) Feedback (d) Control

Solution: The components of the control system ensure smooth functioning of the system to achieve its objective.

Ans. (d)

11. Modifying the software to match changes in the ever-changing environment is called

(a) Adaptive maintenance
(b) Corrective maintenance
(c) Perfective maintenance
(d) Preventive maintenance

Solution: Corrective maintenance refers to modifications initiated by defects in the software.

Perfective maintenance refers to improving processing efficiency or performance, or restructuring the software to improve changeability.

Preventive maintenance refers to prevention of breakdowns.

Ans. (a)