UNIT-1

Introduction

Software Engineering is a framework for building software and is an engineering approach to software development. Software programs can be developed without S/E principles and methodologies but they are indispensable if we want to achieve good quality software in a cost effective manner. Software is defined as:

Instructions + Data Structures + Documents

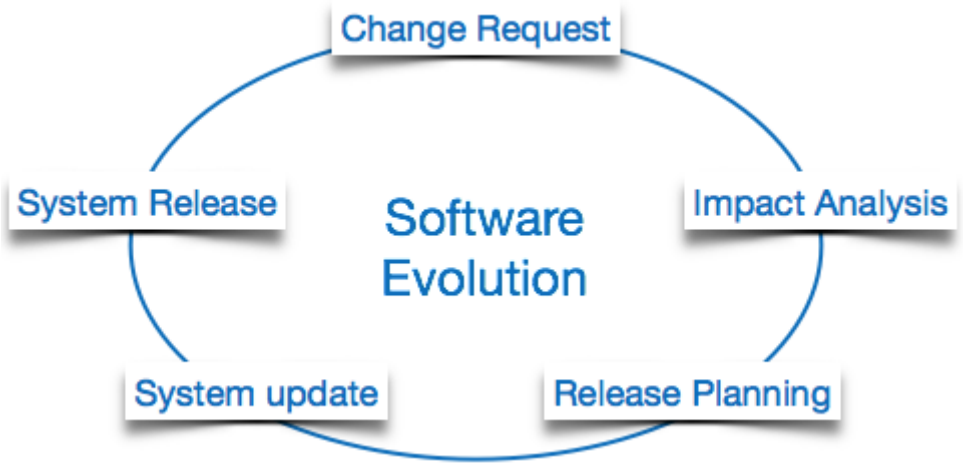
Engineering is the branch of science and technology concerned with the design, building, and use of engines, machines, and structures. It is the application of science, tools and methods to find cost effective solution to simple and complex problems.

SOFTWARE ENGINEERING is defined as a systematic, disciplined and quantifiable approach for the development, operation and maintenance of software.

1.1 Evolution

The process of developing a software product using software engineering principles and methods is referred to as **software evolution.** This includes the initial development of software and its maintenance and updates, till desired software product is developed, which satisfies the expected requirements.

Evolution starts from the requirement gathering process. After which developers create a prototype of the intended software and show it to the users to get their feedback at the early stage of software product development. The users suggest changes, on which several consecutive updates and maintenance keep on changing too. This process changes to the original software, till the desired software is accomplished.



Even after the user has desired software in hand, the advancing technology and the changing requirements force the software product to change accordingly. Re-creating software from

scratch and to go one-on-one with requirement is not feasible. The only feasible and economical solution is to update the existing software so that it matches the latest requirements.

Software Evolution Laws

Lehman has given laws for software evolution. He divided the software into three different categories:

- **S-type (static-type) -** This is a software, which works strictly according to defined specifications and solutions. The solution and the method to achieve it, both are immediately understood before coding. The s-type software is least subjected to changes hence this is the simplest of all. For example, calculator program for mathematical computation.
- **P-type (practical-type) -** This is a software with a collection of procedures. This is defined by exactly what procedures can do. In this software, the specifications can be described but the solution is not obvious instantly. For example, gaming software.
- **E-type (embedded-type) -** This software works closely as the requirement of real-world environment. This software has a high degree of evolution as there are various changes in laws, taxes etc. in the real world situations. For example, online trading software.

1.1.1 Evolution of an Art into an Engineering Discipline

Software engineering principles have evolved over the last sixty years with contributions from numerous researchers and software professionals. Over the years, it has emerged from a pure art to a craft, and finally to an engineering discipline.

The early programmers used an ad hoc programming style. This style of program development is now variously being referred to as exploratory, build and fix, and code and fixes styles. In a build and fix style, a program is quickly developed without making any specification, plan, or design. The different imperfections that are subsequently noticed are fixed.

The exploratory programming style is an informal style in the sense that there are no set rules or recommendations that a programmer has to adhere to—every programmer himself evolves his own software development techniques solely guided by his own intuition, experience, whims, and fancies. The exploratory style comes naturally to all first time programmers. Later in this chapter we point out that except for trivial problems, the exploratory style usually yields poor quality and unmaintainable code and also makes program development very expensive as well as time-consuming.

1.1.2 Evolution Pattern for Engineering Disciplines

If we analyse the evolution of the software development styles over the last sixty years, we can easily notice that it has evolved from an esoteric art form to a craft form, and then has slowly emerged as an engineering discipline. As a matter of fact, this pattern of evolution is

not very different from that seen in other engineering disciplines. Irrespective of whether it is iron making, paper making, software development, or building construction; evolution of technology has followed strikingly similar patterns. This pattern of technology development has schematically been shown in Figure 1.1. It can be seen from Figure 1.1 that every technology in the initial years starts as a form of art. Over time, it graduates to a craft and finally emerges as an engineering discipline. Let us illustrate this fact using an example. Consider the evolution of the iron making technology.

The story of the evolution of the software engineering discipline is not much different. As we have already pointed out, in the early days of programming, there were good programmers and bad programmers. The good programmers knew certain principles (or tricks) that helped them write good programs, which they seldom shared with the bad programmers. Program writing in later years was akin to a craft. Over the next several years, all good principles (or tricks) that were organised into a body of knowledge that forms the discipline of software engineering.

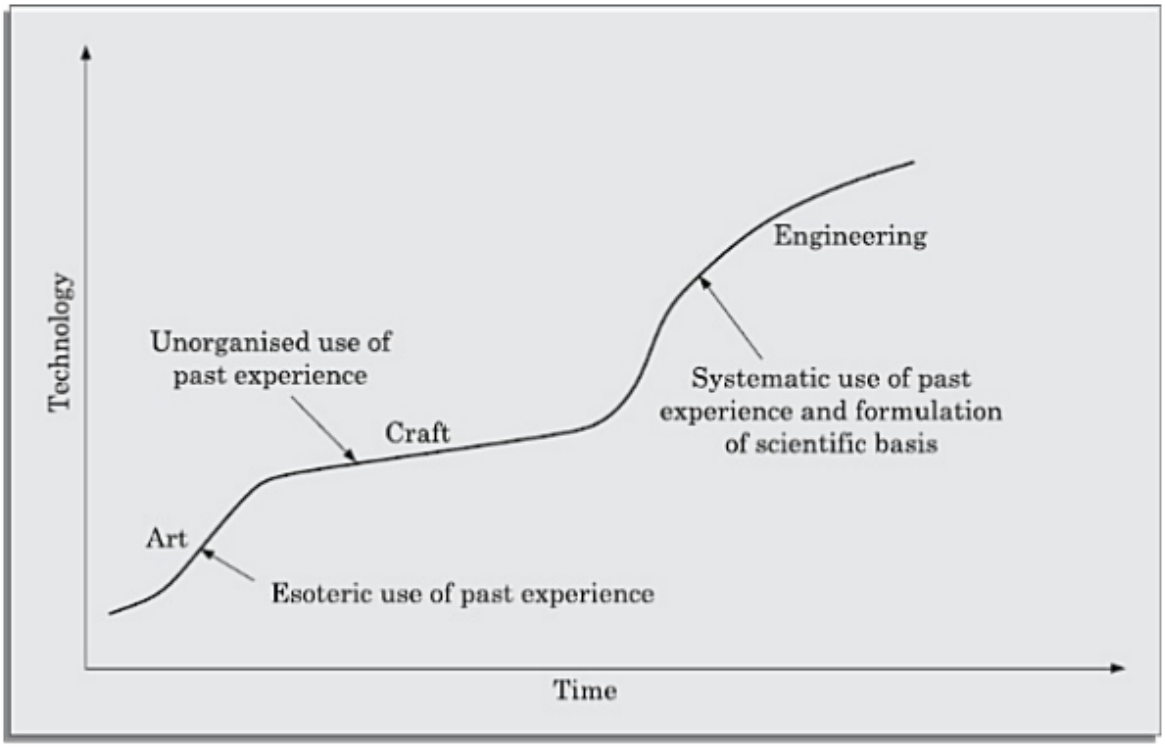


Figure 1.1: Evolution of technology with time.

1.1.3 A Solution to the Software Crisis

At present, software engineering appears to be among the few options that are available to tackle the present software crisis. But, what exactly is the present software crisis? What are its symptoms, causes, and possible solutions? To understand the present software crisis, consider the following facts. The expenses that organisations all over the world are incurring on software purchases as compared to the expenses incurred on hardware purchases have been showing an worrying trend over the years (see Figure 1.2). As can be seen in the figure, organisations are spending increasingly larger portions of their budget on software as

compared to that on hardware. Among all the symptoms of the present software crisis, the trend of increasing software costs is probably the most vexing.

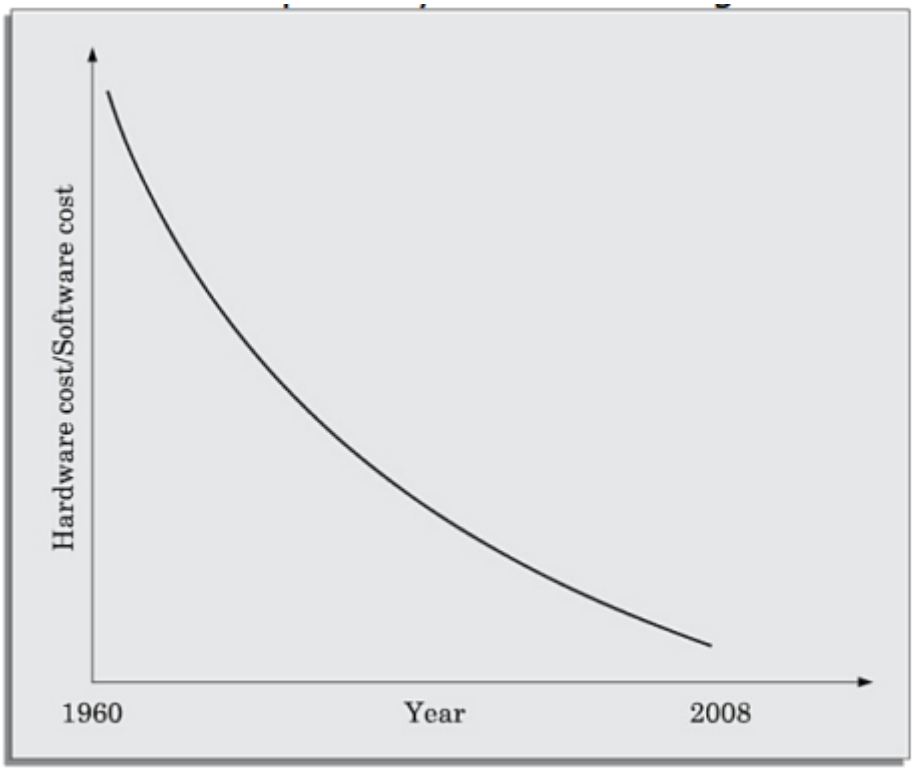


Figure 1.2: Relative changes of hardware and software costs over time.

At present, many organisations are actually spending much more on software than on hardware. If this trend continues, we might soon have a rather amusing scenario. Not long ago, when you bought any hardware product, the essential software that ran on it came free with it. But, unless some sort of revolution happens, in not very distant future, hardware prices would become insignificant compared to software prices—when you buy any software product the hardware on which the software runs would come free with the software.

The symptoms of software crisis are not hard to observe. But, what are the factors that have contributed to the present software crisis? Apparently, there are many factors, the important ones being—rapidly increasing problem size, lack of adequate training in software engineering techniques, increasing skill shortage, and low productivity improvements. What is the remedy? It is believed that a satisfactory solution to the present software crisis can possibly come from a spread of software engineering practices among the developers, coupled with further advancements to the software engineering discipline itself.

1.2 Software Development Projects

Software development projects in software engineering are structured endeavours to design, build, and deploy software solutions to address specific needs or problems. They provide a systematic approach to transforming ideas into tangible, functional software products. These projects leverage software engineering principles to ensure the development process is efficient, scalable, and robust.

The significance of software development projects cannot be overstated. In a world driven by technology, software solutions power industries, streamline operations, and enhance user experiences. These projects drive innovation across diverse sectors, from mobile apps that revolutionize health tracking to e-commerce platforms reshaping the retail landscape.

A software development project is a complex undertaking by two or more persons within the boundaries of time, budget, and staff resources that produces new or enhanced computer code that adds significant business value to a new or existing business process.

Software development projects are important because they:

Drive innovation: Software development projects drive innovation across many sectors, from mobile apps to e-commerce platforms.

Improve user experience: Software development projects can enhance user experiences and boost productivity.

Develop technical skills: Software development projects can help developers hone their programming and problem-solving skills.

Foster collaboration: Software development projects can help foster collaboration among software engineers and cross-functional teams.

1.2.1 Types of Software Development Projects

A software development company is typically structured into a large number of teams that handle various types of software development projects. These software development projects concern the development of either software product or some software service. In the following subsections, we distinguish between these two types of software development projects.

Software products: We all know of a variety of software such as Microsoft's Windows and the Office suite, Oracle DBMS, software accompanying a camcorder or a laser printer, etc. These software's are available off-the-shelf for purchase and are used by a diverse range of customers. These are called generic software products since many users essentially use the same software. These can be purchased off-the-shelf by the customers. When a software development company wishes to develop a generic product, it first determines the features or functionalities that would be useful to a large cross section of users. Based on these, the development team draws up the product specification on its own. Of course, it may base its design discretion on feedbacks collected from a large number of users. Typically, each software product is targeted to some market segment (set of users). For example, Microsoft targets desktops and laptops through its Windows 8 operating system, while it targets high-end mobile handsets through its Windows mobile operating system, and targets servers through its Windows server operating system.

Software services: A software service usually involves either development of a customised software or development of some specific part of software in an outsourced mode.

Customised software is developed according to the specification drawn up by one or at most a few customers. These need to be developed in a short time frame (typically a couple of months), and at the same time the development cost must be low. Usually, a developing company develops customised software by tailoring some of its existing software. For example, when an academic institution wishes to have software that would automate its important activities such as student registration, grading, and fee collection; companies would normally develop such software as a customised product. This means that for developing customised software, the developing company would normally tailor one of its existing software products that it might have developed in the past for some other academic institution.

In a customised software development project, a large part of the software is reused from the code of related software that the company might have already developed. Usually, only a small part of the software that is specific to some client is developed. For example, suppose a software development organisation has developed an academic automation software that automates the student registration, grading, Establishment, hostel and other aspects of an academic institution. When a new educational institution requests for developing software for automation of its activities, a large part of the existing software would be reused. However, a small part of the existing code may be modified to take into account small variations in the required features. For example, software might have been developed for an academic institute that offers only regular residential programs, the educational institute that has now requested for software to automate its activities also offers a distance mode post graduate program where the teaching and sessional evaluations are done by the local centres.

1.2.2 Software Projects Being Undertaken by Indian Companies

Indian software companies have excelled in executing software services projects and have made a name for themselves all over the world. Of late, the Indian companies have slowly started to focus on product development as well. Can you recall the names of a few software products developed by Indian software companies? Let us try to hypothesise the reason for this situation. Generic product development entails certain amount of business risk. A company needs to invest upfront and there is substantial risks concerning whether the investments would turn profitable. Possibly, the Indian companies were risk averse. Till recently, the world-wide sales revenue of software products and services were evenly matched. But, of late the services segment has been growing at a faster pace due to the advent of application service provisioning and cloud computing.

Key Stages in a Software Development Project

1. Project Initiation: Every successful software development project starts with clearly understanding its goals and objectives. Project participants determine the scope of the work during this phase, which also involves identifying the problem that has to be solved and determining its viability. This stage lays the foundation for the entire project, establishing expectations and guiding the subsequent steps.

2. Planning and Design: The project enters the planning and design phase with a clear vision. This is where the project team maps out the project's timeline, allocates resources and budgets, and defines the technical architecture. In software developer projects, meticulous planning ensures smooth and efficient execution.

3. Development: The development phase is the heart of software development projects. Software developers work tirelessly to write code, bring the project to life, and turn concepts into functional applications. Agile practices often come into play, allowing for iterative development, constant feedback, and rapid adaptation to changing requirements.

4. Testing and Quality Assurance: Rigorous testing and quality assurance processes are undertaken before any software is released to users. Software development projects involve various types of testing, including unit testing, integration testing, and user acceptance testing (UAT). This phase ensures the software meets quality standards, functions as intended, and is free from critical defects.

5. Deployment: Once the software is thoroughly tested and refined, it's time for deployment. This phase involves choosing the right deployment strategy—on-premises, cloud-based, or a hybrid model. Deployment is critical, as it marks the transition from development to real-world usage.

6. Project Closure: As the software is deployed and begins serving users, the project closure phase focuses on tying up loose ends. This involves final user acceptance testing, the handover of deliverables and documentation, and a post-implementation review. Lessons learned during this phase contribute to future improvements in software development projects.

Roles and Responsibilities in a Software Development Project

Software development projects involve a symphony of roles, each contributing their unique expertise to bring the project to fruition.

Project Management Roles: Project managers steer the ship, ensuring the project stays on course. Scrum masters facilitate agile practices, while product owners act as the bridge between development teams and stakeholders.

Development Team Roles: Software developers write the code that makes the software function, while UI/UX designers create intuitive and visually appealing user interfaces. QA engineers meticulously test the software to catch bugs and ensure quality.

Supporting Roles: Business analysts define and document requirements, DevOps engineers manage deployment pipelines, and technical writers create comprehensive documentation for users and developers.

Client and Stakeholder Roles

- Client representatives provide project requirements and feedback.
- End users test the software from a usability perspective.
- Project sponsors provide the necessary resources and support.

Some examples of software development projects include:

System software development: This type of project involves creating software that manages and controls computer hardware resources, such as operating systems, device drivers, and utility programs.

Credit card fraud detection: This project uses machine learning to identify suspicious activity in ATM card transactions.

Personalized recommendation system: This project is a good way to develop data science skills.

Recipe organizer: This project involves creating an app that allows users to sort and store their recipes.

1.3 Exploratory Style of Software Development

Exploratory program development style refers to an informal development style or builds and fixes the style in which the programmer uses his own intuition to develop a program rather than making use of the systematic body of knowledge which is categorized under the software engineering discipline. This style of development gives complete freedom to programmers to choose activities which they like to develop software. This dirty program is quickly developed and bugs are fixed whenever it arises.

This style does not offer any rules to start developing any software. The following block diagram will clear some facts relating to this model.

In the below diagram, the first block is the initial briefing by the customer i.e. brief introduction of the problem by the customer. After the briefing, control goes to initial coding i.e. as soon as the developer or programmer knew about the problem he starts coding to develop a working program without considering any kind of requirement analysis. After this, the program will be tested i.e. bugs found and they are getting fixed by programmers. This cycle continues until satisfactory code is not obtained. After finding satisfactory code, development gets completed.

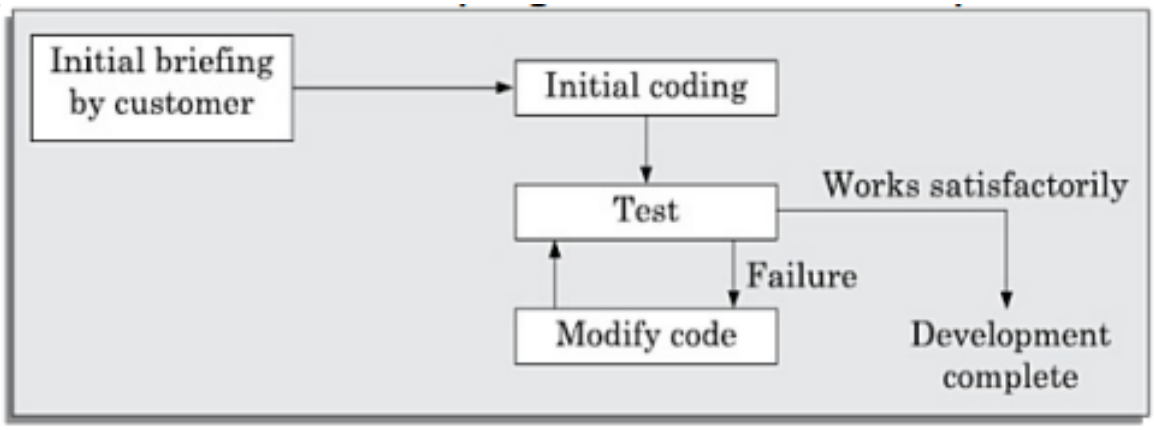


Figure 1.3: Exploratory program development.

Usage

This style of software development is only used for the development of small programs. Nowadays, this style is only used by students in their labs to complete their assignments only. This style is not really used in industries nowadays.

What's wrong with this model?

In an exploratory development scenario, the effort and time required to develop professional software increase with an increase in program size. An increase in development effort and time with problem size has been indicated in the below figure:

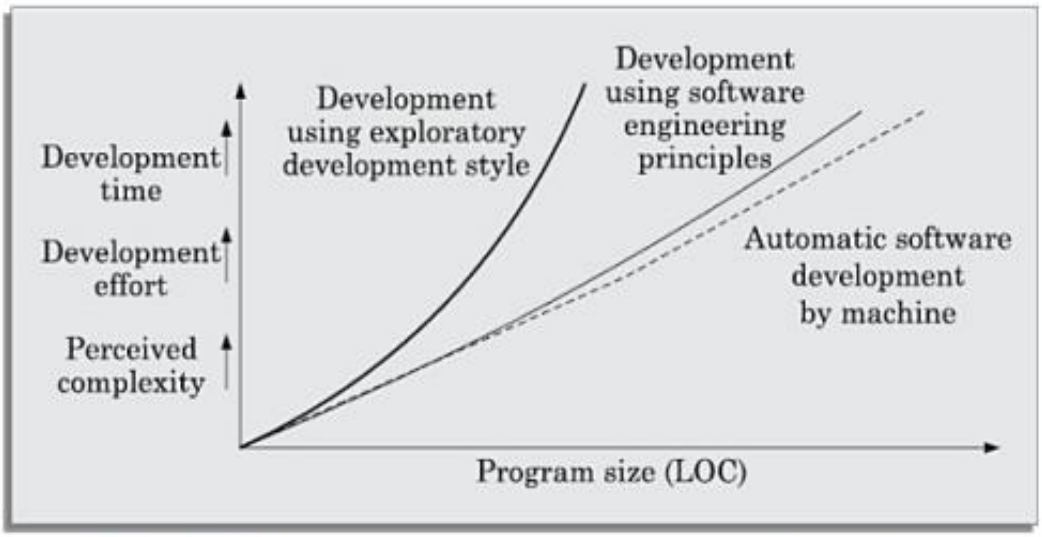


Figure 1.4: Increase in development time and effort with problem size.

In the above figure, the thick line plots represent the case in which the exploratory style is used to develop a program. As program size increases, required effort and time increase almost exponentially. For large problems, it would take too long and cost too much to be practically meaningful to develop a program using the exploratory style of development. The exploratory development approach is said to break down after the size of the program to be developed increases beyond a certain value. The thin solid line is used to represent a case when development is carried out using software engineering principles.

In this case, it becomes possible to solve a problem with effort and time that is almost linear in program size dotted line is used to represent a case when development is carried out by an automated machine. In this case, an increase in effort and time with size would be even closer to a linear increase with size. Exploratory style causes perceived difficulty of a problem to grow exponentially due to human cognitive limitations because whenever the case arises in which a number of independent variables increase in the project then it is beyond the grasping power of an individual and due to this requires more effort.

You may still wonder that when software engineering principles are used, why does the curve not become completely linear? The answer is that it is very difficult to apply decomposition and abstraction principles to completely overcome problem complexity.

The shortcoming of this model:

- Using this model, there is the exponential growth of development time effort and cost with problem size and large-sized software becomes almost impossible using this style of development.
- This style of development results in unmaintainable code because programming without planning always results in unstructured and poor-quality code.
- Also, it is difficult to use this style when there is a proper developing team because in this style every developer uses his own intuition to develop software.

1.3.1 Perceived Problem Complexity: An Interpretation Based on Human Cognition Mechanism

The rapid increase of the perceived complexity of a problem with increase in problem size can be explained from an interpretation of the human cognition mechanism. A simple understanding of the human cognitive mechanism would also give us an insight into why the exploratory style of development leads to an undue increase in the time and effort required to develop a programming solution. It can also explain why it becomes practically infeasible to solve problems larger than a certain size while using an exploratory style; whereas using software engineering principles, the required effort grows almost linearly with size (as indicated by the thin solid line in Figure 1.4).

A small program having just a few variables is within the easy grasp of an individual. As the number of independent variables in the program increases, it quickly exceeds the grasping power of an individual and would require an unduly large effort to master the problem. This outlines a possible reason behind the exponential nature of the effort-size plot (thick line) shown in Figure 1.4. Please note that the situation depicted in Figure 1.4 arises mostly due to the human cognitive limitations. Instead of a human, if a machine could be writing (generating) a program, the slope of the curve would be linear, as the cache size (short-term memory) of a computer is quite large. But, why does the effort-size curve become almost linear when software engineering principles are deployed? This is because software engineering principles extensively use the techniques that are designed specifically to overcome the human cognitive limitations.

1.3.2 Principles Deployed by Software Engineering to Overcome Human Cognitive Limitations

Two important principles that are deployed by software engineering to overcome the problems arising due to human cognitive limitations are abstraction and decomposition.

Abstraction: Abstraction is the simplification of a problem by focusing on only one aspect of the problem while omitting all other aspects.

When using the principle of abstraction to understand a complex problem, we focus our attention on only one or two specific aspects of the problem and ignore the rest. Whenever

we omit some details of a problem to construct an abstraction, we construct a model of the problem. In everyday life, we use the principle of abstraction frequently to understand a problem or to assess a situation.

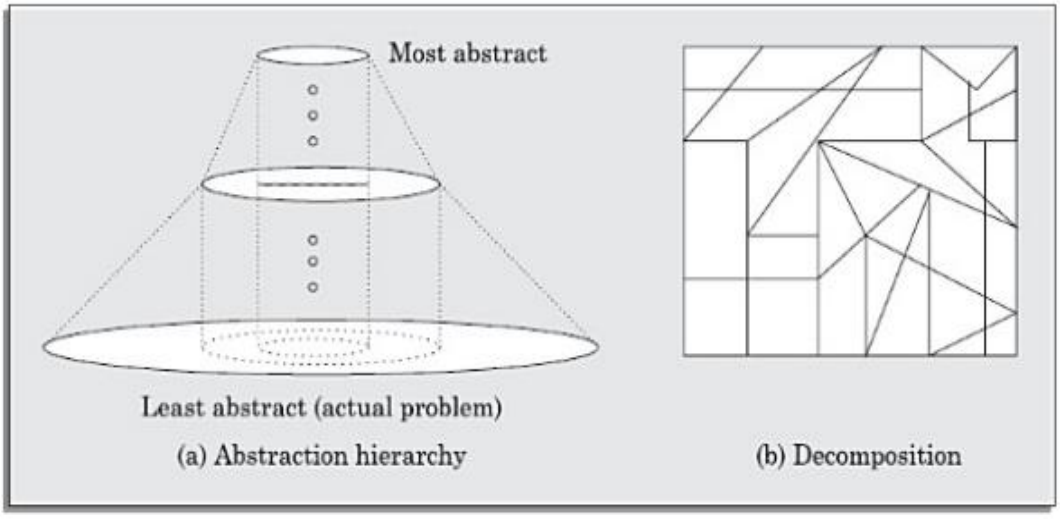


Figure 1.6: Schematic representation.

A single level of abstraction can be sufficient for rather simple problems. However, more complex problems would need to be modelled as a hierarchy of abstractions. A schematic representation of an abstraction hierarchy has been shown in Figure 1.6(a). The most abstract representation would have only a few items and would be the easiest to understand. After one understands the simplest representation, one would try to understand the next level of abstraction where at most five or seven new information are added and so on until the lowest level is understood. By the time, one reaches the lowest level; he would have mastered the entire problem.

Decomposition: The decomposition principle advocates decomposing the problem into many small independent parts. The small parts are then taken up one by one and solved separately. The idea is that each small part would be easy to grasp and understand and can be easily solved. The full problem is solved when all the parts are solved.

A popular way to demonstrate the decomposition principle is by trying to break a large bunch of sticks tied together and then breaking them individually. Figure 1.6(b) shows the decomposition o f a large problem into many small parts. However, it is very important to understand that any arbitrary decomposition of a problem into small parts would not help. The different parts after decomposition should be more or less independent of each other. That is, to solve one part you should not have to refer and understand other parts. If to solve one part you would have to understand other parts, then this would boil down to understanding all the parts together. This would effectively reduce the problem to the original problem before decomposition (the case when all the sticks tied together). Therefore, it is not sufficient to just decompose the problem in any way, but the decomposition should be such that the different decomposed parts must be more or less independent of each other.

1.4 Emergence of Software Engineering

Software engineering emerged in the late 1960s as a new engineering discipline concerned with all aspects pertaining to software production. It encompasses concepts, principles, theories, techniques and tools that can be used for developing high-quality professional software.

The emergence of software engineering refers to the development of the discipline of software engineering in the late 1960s:

Background

The field of software engineering emerged from the need for better quality assurance and reliability in software development. The first software engineering conference was held in 1968 by NATO to address these issues.

Principles

The conference brought together international experts who agreed that the systematic approach of physical world engineering should be applied to software development.

Early practices: Early software engineering practices included:

- Adopting formal, rigid approaches like the waterfall method to counteract haphazard programming
- Using flow charts to design control flow structures and understand program logic

1.4.1 Early Computer Programming:

As we know that in the early 1950s, computers were slow and expensive. Though the programs at that time were very small in size, these computers took considerable time to process them. They relied on assembly language which was specific to computer architecture. Thus, developing a program required lot of effort. Every programmer used his own style to develop the programs.

1.4.2 High Level Language Programming

With the introduction of semiconductor technology, the computers became smaller, faster, cheaper, and reliable than their predecessors. One of the major developments includes the progress from assembly language to high-level languages. Early high level programming languages such as COBOL and FORTRAN came into existence. As a result, the programming became easier and thus, increased the productivity of the programmers. However, still the programs were limited in size and the programmers developed programs using their own style and experience.

1.4.3 Control Flow Based Design

With the advent of powerful machines and high level languages, the usage of computers grew rapidly: In addition, the nature of programs also changed from simple to complex. The

increased size and the complexity could not be managed by individual style. It was analyzed that clarity of control flow (the sequence in which the program's instructions are executed) is of great importance. To help the programmer to design programs having good control flow structure, **flowcharting technique** was developed. In flowcharting technique, the algorithm is represented using flowcharts. A **flowchart** is a graphical representation that depicts the sequence of operations to be carried out to solve a given problem.

Note that having more GOTO constructs in the flowchart makes the control flow messy, which makes it difficult to understand and debug. In order to provide clarity of control flow, the use of GOTO constructs in flowcharts should be avoided and **structured constructs-decision,** sequence, and loop-should be used to develop **structured flowcharts.** The decision structures are used for conditional execution of statements (for example, if statement). The sequence structures are used for the sequentially executed statements. The loop structures are used for performing some repetitive tasks in the program. The use of structured constructs formed the basis of the **structured programming** methodology.

1.4.4 Data Structure-oriented Design

Using data structure-oriented design techniques, first a program's data structures are designed. The code structure is designed based on the data structure. In the next step, the program design is derived from the data structure. An example of a data structure-oriented design technique is the Jackson's Structured Programming (JSP) technique developed by Michael Jackson [1975]. In JSP methodology, a program's data structure is first designed using the notations for sequence, selection, and iteration. The JSP methodology provides an interesting technique to derive the program structure from its data structure representation. Several other data structure-based design techniques were also developed. Some of these techniques became very popular and were extensively used.

1.4.5 Data-Flow Oriented Design

With the introduction of very Large Scale Integrated circuits (VLSI), the computers became more powerful and faster. As a result, various significant developments like networking and GUIs came into being. Clearly, the complexity of software could not be dealt using control flow based design. Thus, a new technique, namely, **data-flow-oriented** technique came into existence. In this technique, the flow of data through business functions or processes is represented using **Data-flow Diagram (DFD). IEEE** defines a data-flow diagram (also known as **bubble chart** and **work-flow diagram)** as 'a diagram that depicts data sources, data sinks, data storage, and processes performed on data as nodes, and logical flow of data as links between the nodes.'

1.4.6 Object Oriented Design

Object-oriented design technique has revolutionized the process of software development. It not only includes the best features of structured programming but also some new and powerful features such as encapsulation, abstraction, inheritance, and polymorphism. These new features have tremendously helped in the development of well-designed and high-quality

software. Object-oriented techniques are widely used these days as they allow reusability of the code. They lead to faster software development and high-quality programs. Moreover, they are easier to adapt and scale, that is, large systems can be created by assembling reusable subsystems.

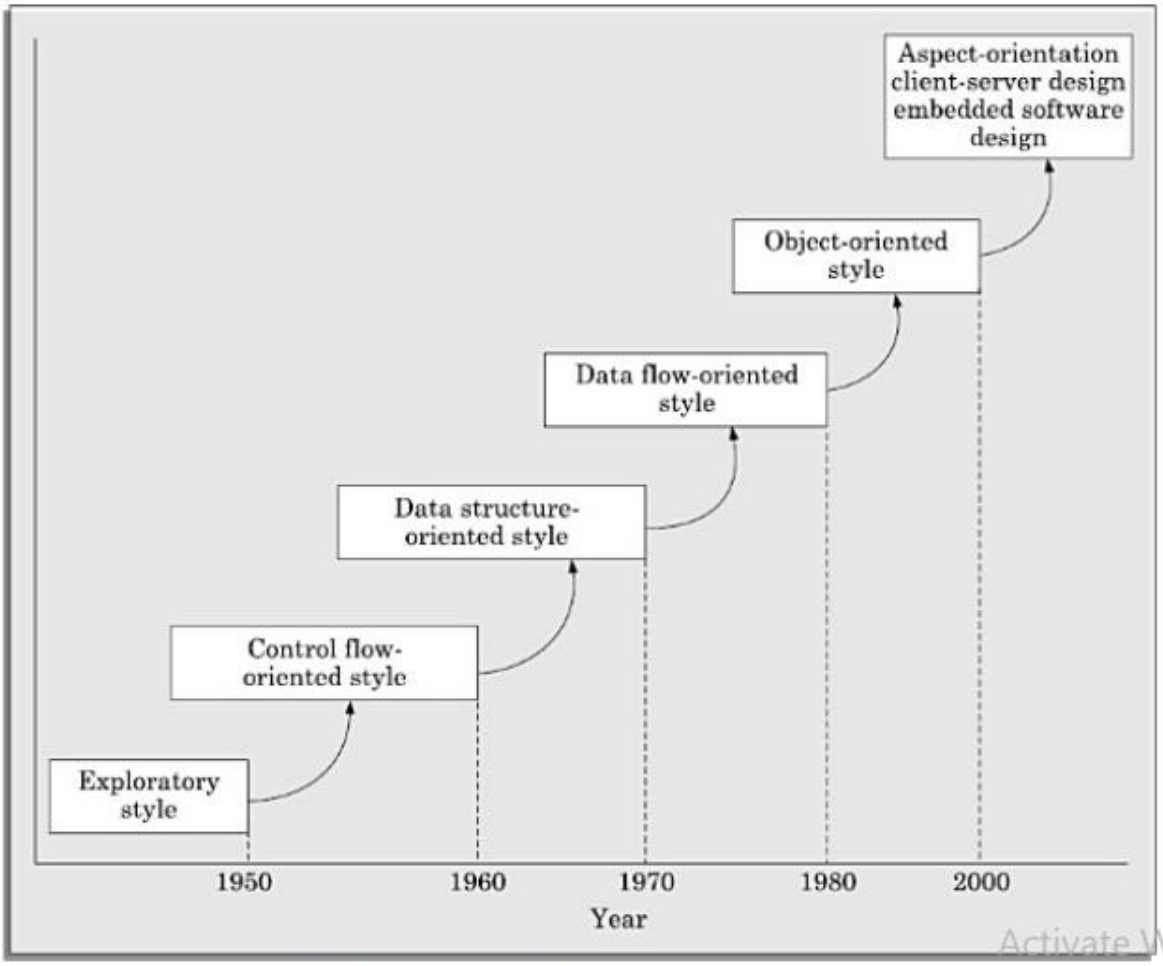


Figure 1.12: Evolution of software design techniques.

1.4.7 Other Developments

Over time, more flexible software engineering methods emerged to counteract the shortcomings of the waterfall method. In the 1990s, software engineers began to focus on the human element, including:

- Graphical user interfaces
- Open-source code
- Human-computer interaction
- Agile software engineering methods

Software engineering is the process of developing, testing, and deploying computer applications to solve real-world problems. It emphasizes a systematic, disciplined approach to software development.

Software engineering discipline is the result of advancement in the field of technology. In this section, we will discuss various innovations and technologies that led to the emergence of software engineering discipline.

1.5 Notable Changes in Software Development Practices

Significant changes in software development practices include a shift towards Agile methodologies, increased adoption of cloud computing, greater emphasis on DevOps for collaboration and automation, the rise of microservices architecture, growing use of AI and machine learning, and a focus on user experience design with tools like low-code and no-code platforms, all aimed at faster development cycles and adaptability to changing requirements.

Key aspects of these changes:

Agile Development: This flexible and adaptable methodology has become popular with businesses of all sizes. It emphasizes collaboration between teams and rapid releases, and can help improve software quality and user experience.

Cloud Computing: Enables scalable and flexible infrastructure, allowing developers to focus on building applications without managing hardware, leading to faster deployment and cost optimization.

DevOps: This approach focuses on improving collaboration between development and operations teams. It requires software engineers to be familiar with monitoring tools, infrastructure automation, and continuous integration/continuous deployment (CI/CD) practices.

Microservices Architecture: This architecture allows modules to be built, managed, and changed without affecting other modules. This makes microservices easier to scale and reuse, and can save time and money when problems occur.

Breaks down large applications into smaller, independent services that can be developed, deployed, and scaled independently, improving maintainability and agility.

Artificial Intelligence (AI) and Machine Learning (ML): Artificial Intelligence (AI) and Machine Learning (ML) are transformative technologies used to automate repetitive tasks, enhance decision-making, and improve software functionalities. They leverage predictive analytics to analyze data, identify patterns, and make informed predictions, enabling smarter and more efficient systems across various domains.

Low-code and No-code Platforms: Low-code and no-code platforms enable users, even with minimal technical expertise, to create applications using intuitive visual interfaces and drag-and-drop tools. These platforms streamline the development process, significantly reducing the time and effort required to build and deploy applications.

User Experience (UX) Design: User Experience (UX) Design emphasizes user-centric design principles to craft intuitive, accessible, and engaging interfaces. It prioritizes

understanding user needs, behaviors, and preferences, ensuring seamless interactions and improved satisfaction. This approach fosters usability, enhancing both functionality and aesthetics to create meaningful digital experiences.

Continuous Integration and Continuous Delivery (CI/CD): CI/CD automates the software development lifecycle by streamlining the processes of building, testing, and deploying code. This approach enables faster feedback loops, reduces manual errors, and facilitates frequent, reliable software releases, enhancing development efficiency and product quality.

Other notable changes:

Remote Work: The adoption of remote work in development has surged, driven by advancements in communication tools, cloud-based platforms, and collaborative technologies. Flexible work arrangements allow teams to operate across diverse geographies, improving work-life balance and access to global talent. This shift has redefined traditional workflows, emphasizing asynchronous collaboration and productivity-focused practices while reducing overhead costs for businesses.

Cybersecurity Concerns: With the increasing frequency and sophistication of cyber threats, organizations are prioritizing the integration of robust security measures throughout the software development lifecycle. DevSecOps emphasizes embedding security practices early in the development process, ensuring vulnerabilities are addressed proactively. This approach fosters collaboration among development, security, and operations teams, enabling faster delivery of secure, resilient software while reducing the risk of costly breaches.

Emerging Programming Languages: Emerging programming languages like “Rust” are gaining traction for their unique capabilities tailored to specific use cases. Rust, known for its memory safety without a garbage collector, excels in system-level programming, embedded systems, and performance-critical applications. Its focus on concurrency and safety makes it a preferred choice for developers building secure, efficient, and reliable software. Adoption is driven by its ability to address issues like memory leaks and data races, common in older languages like C and C++, while maintaining high performance. Rust’s growing ecosystem and developer-friendly features further encourage its use in industries ranging from cybersecurity to web development.

Internet of Things (IoT): The Internet of Things (IoT) has revolutionized software development by enabling seamless connectivity among devices, creating both opportunities and challenges. IoT software powers smart devices to collect, process, and share data, unlocking innovations in healthcare, transportation, and smart homes. However, challenges such as data security, interoperability, and energy efficiency require robust solutions. IoT development emphasizes scalable architectures, real-time processing, and enhanced cybersecurity to meet these demands while fostering smarter and more integrated systems.

1.6 Computer Systems Engineering: Computer systems engineering (CSYE) is a multidisciplinary fielding that combine’s computer science, engineering, and mathematical analysis to design, develop, and maintain computer systems:

Software and hardware

CSYE involves designing, developing, and maintaining both software and hardware components.

System architecture

CSYE involves designing the overall structure of a computer system, including the arrangement of hardware components and the communication between them.

Problem solving

CSYE involves applying theories and principles of computing, mathematics, science, and engineering to solve technical problems.

CSYE is in demand in many industries, including: government, manufacturing, transportation, and telecommunications.

CSYE practitioners are responsible for:

- Designing new applications of computers and other devices
- Ensuring that computer systems are reliable and secure
- Meeting the demands of today's technology-driven world

The general model of systems engineering is shown schematically in Figure 1.13. One of the important stages in systems engineering i s the stage in which decision is made regarding the parts of the problems that are to be implemented in hardware and the ones that would be implemented in software. This has been represented by the box captioned hardware-software partitioning in Figure 1.13. While partitioning the functions between hardware and software, several trade-offs such as flexibility, cost, speed of operation, etc., need to be considered. The functionality implemented in hardware run faster.

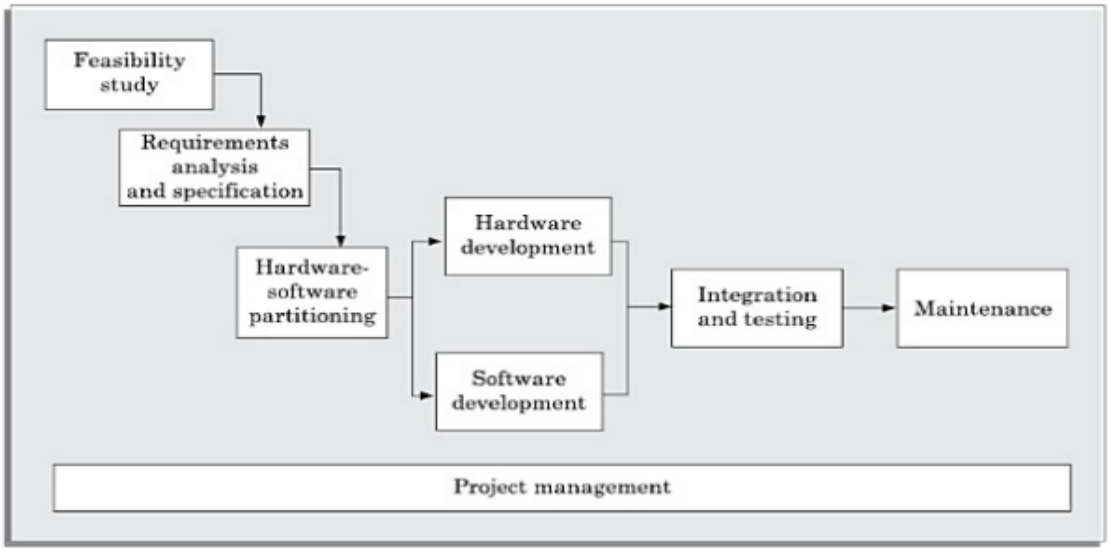


Figure 1.13: Computer systems engineering.

On the other hand, functionalities implemented in software is easier to extend. Further, it is difficult to implement complex functions in hardware. Also, functions implemented in hardware incur extra space, weight, manufacturing cost, and power overhead.

After the hardware-software partitioning stage, development of hardware and software are carried out concurrently (shown as concurrent branches in Figure 1.13). In system engineering, testing the software during development becomes a tricky issue, the hardware on which the software would run and tested would still be under development—remember that the hardware and the software are being developed at the same time. To test the software during development, it usually becomes necessary to develop simulators that mimic the features of the hardware being developed. The software is tested using these simulators. Once both hardware and software development are complete, these are integrated and tested. The project management activity is required throughout the duration of system development as shown in Figure 1.13. In this text, we have confined our attention to software engineering only.

Key aspects of Computer Systems Engineering:

Hardware Design: Designing and optimizing components like processors, memory, and input/output devices.

Software Development: Creating operating systems, applications, and drivers to interact with the hardware.

Network Architecture: Designing and managing computer networks to facilitate data communication.

System Integration: Combining different hardware and software components to create a functional system.

Performance Optimization: Analyzing and improving the efficiency of computer systems.

Applications of Computer Systems Engineering:

Embedded Systems:

Designing computer systems for specialized devices like smart home appliances, medical equipment, and industrial controllers.

High-Performance Computing:

Developing powerful computer systems for complex simulations and data analysis.

Cloud Computing:

Designing and managing cloud infrastructure for scalable computing services.

Robotics:

Integrating computer systems into robotic systems for autonomous operation.

Software Life Cycle Models

1.7 Basic concepts

A software life cycle model refers to a structured process that outlines the different stages involved in developing software, from initial planning and requirement analysis to design, development, testing, deployment, and maintenance, with each model offering a distinct approach to managing these phases depending on the project needs and complexity; common models include Waterfall, Agile, Spiral, V-model, and Iterative/Incremental, each with its own advantages and limitations.

Key Concepts in Software Life Cycle Models:

Requirement Analysis: Defining the exact functionalities and features needed in the software, including user needs and system constraints.

Design: Creating a blueprint for the software architecture, including system components, user interface, and data flow.

Development: Implementing the software code based on the design specifications.

Testing: Thoroughly evaluating the software to identify and fix bugs or errors, ensuring quality and functionality.

Deployment: Distributing the completed software to the intended users.

Maintenance: Addressing issues that arise after deployment, including bug fixes, updates, and enhancements.

Common Software Life Cycle Models:

Waterfall Model: A linear process where each phase is completed sequentially before moving to the next, with strict documentation and minimal flexibility.

Agile Model: Emphasizes iterative development with short cycles ("sprints"), frequent feedback from users, and adaptability to changing requirements.

Spiral Model: A risk-driven approach with repeated cycles of prototyping, evaluation, and refinement, suitable for complex projects with high uncertainty.

V-Model: A verification and validation model where each development phase is directly linked to a corresponding testing phase, ensuring quality throughout the process.

Iterative Model: A gradual development process where initial functionality is built and refined through multiple iterations, incorporating feedback along the way.

Choosing the Right Model

Project complexity: For simple projects, Waterfall may be suitable, while complex projects might benefit from Agile or Spiral models.

Requirement stability: If requirements are well-defined and unlikely to change, Waterfall can be effective, while Agile is better for projects with evolving needs.

Client involvement: Agile encourages high client involvement with frequent feedback loops.

1.8 Waterfall Model and its Extensions

Winston Royce introduced the Waterfall Model in 1970. The Waterfall Model was the first Process Model to be introduced. It is also referred to as a linear-sequential life cycle model. It is very simple to understand and use. In a waterfall model, each phase must be completed before the next phase can begin and there is no overlapping in the phases.

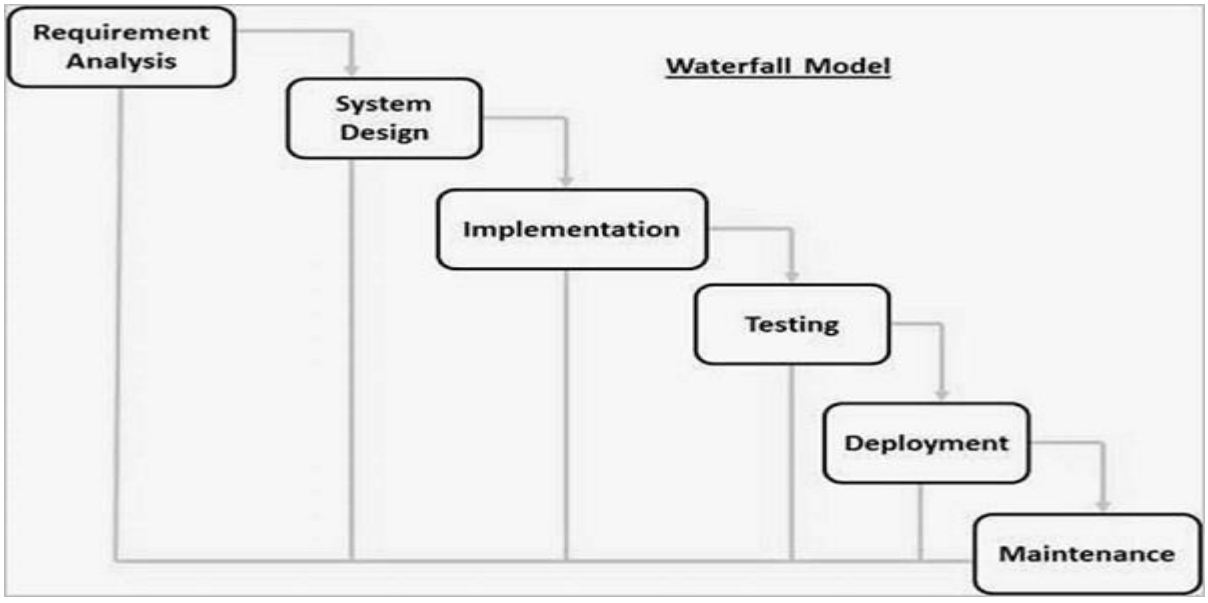
The Waterfall model is the earliest SDLC approach that was used for software development.

The waterfall Model illustrates the software development process in a linear sequential flow. This means that any phase in the development process begins only if the previous phase is complete. In this waterfall model, the phases do not overlap.

Waterfall Model - Design

Waterfall approach was first SDLC Model to be used widely in Software Engineering to ensure success of the project. In "The Waterfall" approach, the whole process of software development is divided into separate phases. In this Waterfall model, typically, the outcome of one phase acts as the input for the next phase sequentially.

The following illustration is a representation of the different phases of the Waterfall Model.



The sequential phases in Waterfall model are:

Requirement Gathering and analysis: All possible requirements of the system to be developed are captured in this phase and documented in a requirement specification document.

System Design: The requirement specifications from first phase are studied in this phase and the system design is prepared. This system design helps in specifying hardware and system requirements and helps in defining the overall system architecture.

Implementation: With inputs from the system design, the system is first developed in small programs called units, which are integrated in the next phase. Each unit is developed and tested for its functionality, which is referred to as Unit Testing.

Integration and Testing: All the units developed in the implementation phase are integrated into a system after testing of each unit. Post integration the entire system is tested for any faults and failures.

Deployment of system: Once the functional and non-functional testing is done; the product is deployed in the customer environment or released into the market.

Maintenance: There are some issues which come up in the client environment. To fix those issues, patches are released. Also to enhance the product some better versions are released. Maintenance is done to deliver these changes in the customer environment.

All these phases are cascaded to each other in which progress is seen as flowing steadily downwards (like a waterfall) through the phases. The next phase is started only after the defined set of goals are achieved for previous phase and it is signed off, so the name "Waterfall Model". In this model, phases do not overlap.

When to use SDLC Waterfall Model?

Some Circumstances where the use of the Waterfall model is most suited are:

- When the requirements are constant and not changed regularly.
- A project is short
- The situation is calm
- Where the tools and technology used is consistent and is not changing
- When resources are well prepared and are available to use.

Advantages of the Waterfall Model

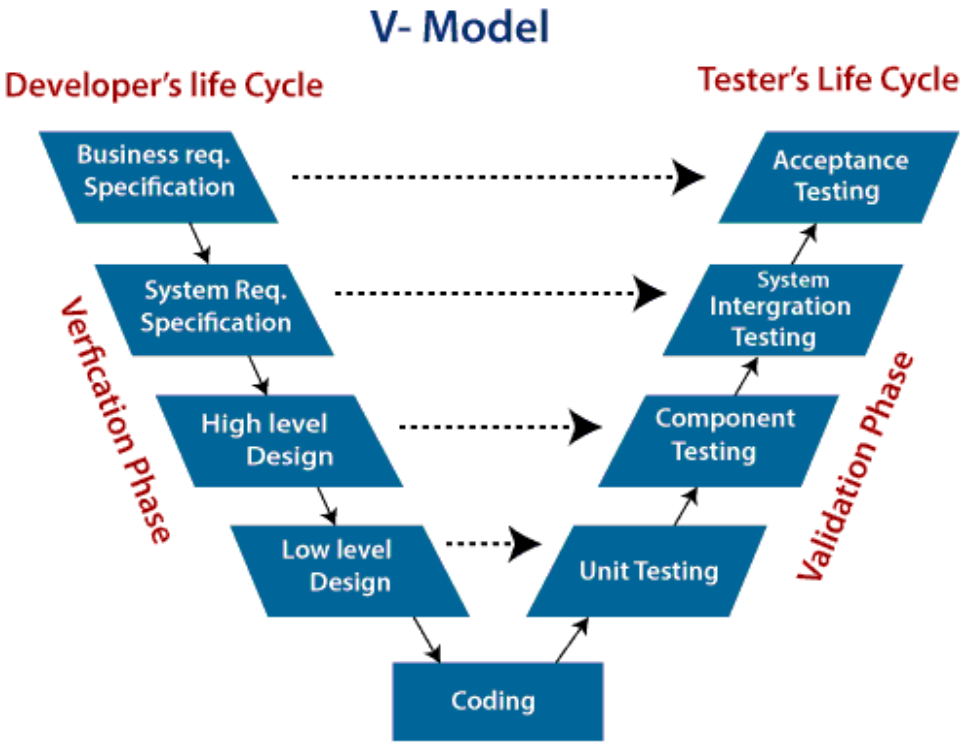
- Simple and easy to understand and use
- Easy to manage due to the rigidity of the model. Each phase has specific deliverables and a review process.
- Phases are processed and completed one at a time.
- Works well for smaller projects where requirements are very well understood.
- Clearly defined stages.

- Well understood milestones.
- Easy to arrange tasks.
- Process and results are well documented.

Waterfall is a sequential, linear approach, while V-Model is an extension of Waterfall that emphasizes testing at each stage. V-Model is more adaptable to changes compared to traditional Waterfall due to its focus on testing and verification.

V-Model

V-Model also referred to as the Verification and Validation Model. In this, each phase of SDLC must complete before the next phase starts. It follows a sequential design process same as the waterfall model. Testing of the device is planned in parallel with a corresponding stage of development.



Verification: It involves a static analysis method (review) done without executing code. It is the process of evaluation of the product development process to find whether specified requirements meet.

Validation: It involves dynamic analysis method (functional, non-functional), testing is done by executing code. Validation is the process to classify the software after the completion of the development process to determine whether the software meets the customer expectations and requirements.

So V-Model contains Verification phases on one side of the Validation phases on the other side. Verification and Validation process is joined by coding phase in V-shape. Thus it is known as V-Model.

There are the various phases of Verification Phase of V-model:

Business requirement analysis: This is the first step where product requirements understood from the customer's side. This phase contains detailed communication to understand customer's expectations and exact requirements.

System Design: In this stage system engineers analyze and interpret the business of the proposed system by studying the user requirements document.

Architecture Design: The baseline in selecting the architecture is that it should understand all which typically consists of the list of modules, brief functionality of each module, their interface relationships, dependencies, database tables, architecture diagrams, technology detail, etc. The integration testing model is carried out in a particular phase.

Module Design: In the module design phase, the system breaks down into small modules. The detailed design of the modules is specified, which is known as Low-Level Design

Coding Phase: After designing, the coding phase is started. Based on the requirements, a suitable programming language is decided. There are some guidelines and standards for coding. Before checking in the repository, the final build is optimized for better performance, and the code goes through many code reviews to check the performance.

There are the various phases of Validation Phase of V-model:

Unit Testing: In the V-Model, Unit Test Plans (UTPs) are developed during the module design phase. These UTPs are executed to eliminate errors at code level or unit level. A unit is the smallest entity which can independently exist, e.g., a program module. Unit testing verifies that the smallest entity can function correctly when isolated from the rest of the codes/ units.

Integration Testing: Integration Test Plans are developed during the Architectural Design Phase. These tests verify that groups created and tested independently can coexist and communicate among themselves.

System Testing: System Tests Plans are developed during System Design Phase. Unlike Unit and Integration Test Plans, System Tests Plans are composed by the client’s business team. System Test ensures that expectations from an application developer are met.

Acceptance Testing: Acceptance testing is related to the business requirement analysis part. It includes testing the software product in user atmosphere. Acceptance tests reveal the compatibility problems with the different systems, which is available within the user atmosphere. It conjointly discovers the non-functional problems like load and performance defects within the real user atmosphere.

When to use V-Model?

- When the requirement is well defined and not ambiguous.

- The V-shaped model should be used for small to medium-sized projects where requirements are clearly defined and fixed.
- The V-shaped model should be chosen when sample technical resources are available with essential technical expertise.

Advantage of V-Model:

- Easy to Understand.
- Testing Methods like planning, test designing happens well before coding.
- This saves a lot of time. Hence a higher chance of success over the waterfall model.
- Avoids the downward flow of the defects.
- Works well for small plans where requirements are easily understood.

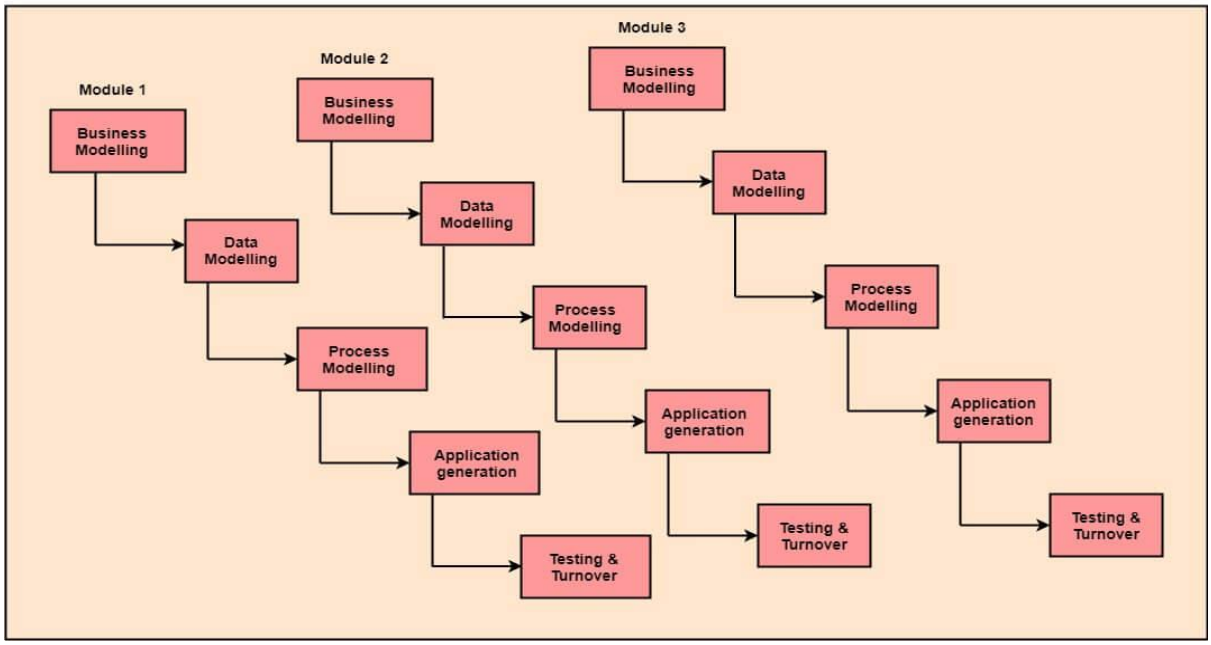
1.9 Rapid Application Development (RAD) Model

RAD is a linear sequential software development process model that emphasizes a concise development cycle using an element based construction approach. If the requirements are well understood and described, and the project scope is a constraint, the RAD process enables a development team to create a fully functional system within a concise time period.

RAD (Rapid Application Development) is a concept that products can be developed faster and of higher quality through:

- Gathering requirements using workshops or focus groups
- Prototyping and early, reiterative user testing of designs
- The re-use of software components
- A rigidly paced schedule that refers design improvements to the next product version
- Less formality in reviews and other team communication

Fig: RAD Model



The various phases of RAD are as follows:

1.Business Modelling: The information flow among business functions is defined by answering questions like what data drives the business process, what data is generated, who generates it, where does the information go, who process it and so on.

2. Data Modelling: The data collected from business modeling is refined into a set of data objects (entities) that are needed to support the business. The attributes (character of each entity) are identified, and the relation between these data objects (entities) is defined.

3. Process Modelling: The information object defined in the data modeling phase are transformed to achieve the data flow necessary to implement a business function. Processing descriptions are created for adding, modifying, deleting, or retrieving a data object.

4. Application Generation: Automated tools are used to facilitate construction of the software; even they use the 4th GL techniques.

5. Testing & Turnover: Many of the programming components have already been tested since RAD emphasis reuse. This reduces the overall testing time. But the new part must be tested, and all interfaces must be fully exercised.

When to use RAD Model?

- When the system should need to create the project that modularizes in a short span time (2-3 months).
- When the requirements are well-known.
- When the technical risk is limited.
- When there's a necessity to make a system, which modularized in 2-3 months of period.
- It should be used only if the budget allows the use of automatic code generating tools.

Advantage of RAD Model

- This model is flexible for change.
- In this model, changes are adoptable.
- Each phase in RAD brings highest priority functionality to the customer.
- It reduced development time.
- It increases the reusability of features.

1.10 Agile Development Model

The meaning of Agile is swift or versatile. “Agile process model" refers to a software development approach based on iterative development. Agile methods break tasks into smaller iterations, or parts do not directly involve long term planning. The project scope and requirements are laid down at the beginning of the development process. Plans regarding the number of iterations, the duration and the scope of each iteration are clearly defined in advance.

Each iteration is considered as a short time "frame" in the agile process model, which typically lasts from one to four weeks. The division of the entire project into smaller parts helps to minimize the project risk and to reduce the overall project delivery time requirements. Each iteration involves a team working through a full software development life cycle including planning, requirements analysis, design, coding, and testing before a working product is demonstrated to the client.

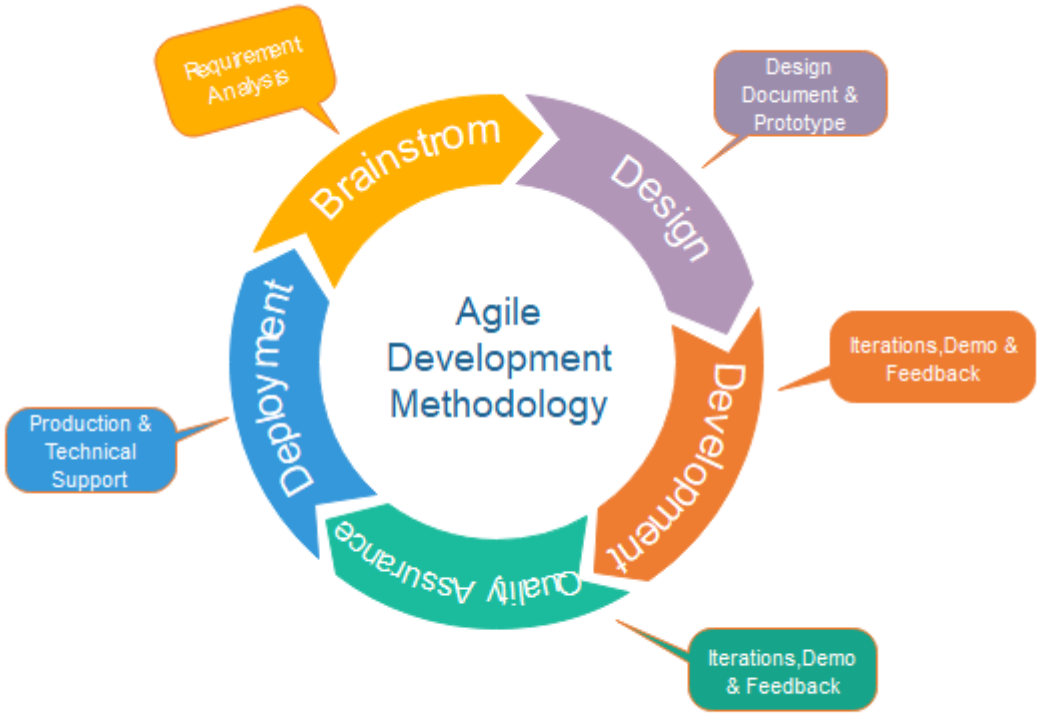


Fig. Agile Model

Phases of Agile Model:

Following are the phases in the agile model are as follows:

1. Requirements gathering
2. Design the requirements
3. Construction/ iteration
4. Testing/ Quality assurance
5. Deployment
6. Feedback

1. Requirements gathering: In this phase, you must define the requirements. You should explain business opportunities and plan the time and effort needed to build the project. Based on this information, you can evaluate technical and economic feasibility.

2. Design the requirements: When you have identified the project, work with stakeholders to define requirements. You can use the user flow diagram or the high-level UML diagram to show the work of new features and show how it will apply to your existing system.

3. Construction/ iteration: When the team defines the requirements, the work begins. Designers and developers start working on their project, which aims to deploy a working product. The product will undergo various stages of improvement, so it includes simple, minimal functionality.

4. Testing: In this phase, the Quality Assurance team examines the product's performance and looks for the bug.

5. Deployment: In this phase, the team issues a product for the user's work environment.

6. Feedback: After releasing the product, the last step is feedback. In this, the team receives feedback about the product and works through the feedback.

Agile Testing Methods:

- Scrum
- Crystal
- Dynamic Software Development Method(DSDM)
- Feature Driven Development(FDD)
- Lean Software Development
- EXtreme Programming(XP)

When to use the Agile Model?

- When frequent changes are required.
- When a highly qualified and experienced team is available.
- When a customer is ready to have a meeting with a software team all the time.
- When project size is small.

Advantages of Agile Method:

- Frequent Delivery
- Face-to-Face Communication with clients.
- Efficient design and fulfils the business requirement.
- Anytime changes are acceptable.
- It reduces total development time.

1.11 Spiral model

The spiral model, initially proposed by Boehm, is an evolutionary software process model that couples the iterative feature of prototyping with the controlled and systematic aspects of the linear sequential model. It implements the potential for rapid development of new versions of the software. Using the spiral model, the software is developed in a series of incremental releases. During the early iterations, the additional release may be a paper model or prototype. During later iterations, more and more complete versions of the engineered system are produced.

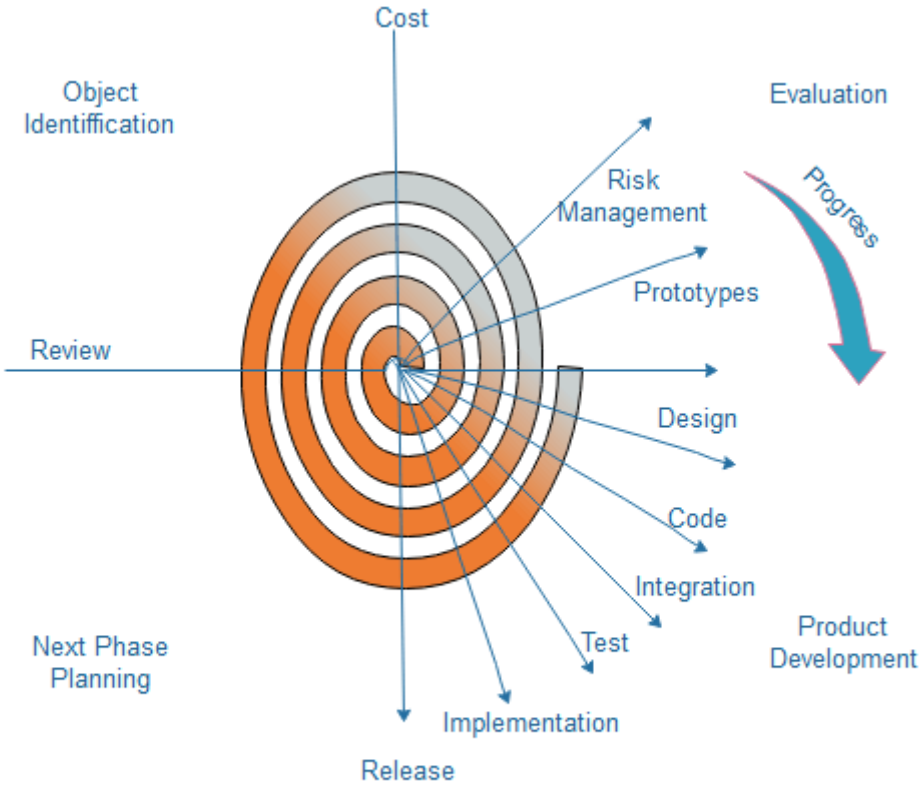


Fig. Spiral Model

Each cycle in the spiral is divided into four parts:

Objective setting: Each cycle in the spiral starts with the identification of purpose for that cycle, the various alternatives that are possible for achieving the targets, and the constraints that exists.

Risk Assessment and reduction: The next phase in the cycle is to calculate these various alternatives based on the goals and constraints. The focus of evaluation in this stage is located on the risk perception for the project.

Development and validation: The next phase is to develop strategies that resolve uncertainties and risks. This process may include activities such as benchmarking, simulation, and prototyping.

Planning: Finally, the next step is planned. The project is reviewed, and a choice made whether to continue with a further period of the spiral. If it is determined to keep, plans are drawn up for the next step of the project.

The development phase depends on the remaining risks. For example, if performance or user-interface risks are treated more essential than the program development risks, the next phase may be an evolutionary development that includes developing a more detailed prototype for solving the risks.

The risk-driven feature of the spiral model allows it to accommodate any mixture of a specification-oriented, prototype-oriented, simulation-oriented, or another type of approach. An essential element of the model is that each period of the spiral is completed by a review that includes all the products developed during that cycle, including plans for the next cycle. The spiral model works for development as well as enhancement projects.

When to use Spiral Model?

- When deliverance is required to be frequent.
- When the project is large
- When requirements are unclear and complex
- When changes may require at any time
- Large and high budget projects

Advantages of Spiral Model

- High amount of risk analysis
- Useful for large and mission-critical projects