Chapter 1

Software & Software Engineering

- Computer software remains the paramount technology globally. With its increasing significance, the software community consistently endeavours to create technologies that streamline the process of building and sustaining top-notch computer programs, aiming to make it simpler, quicker, and more cost-effective.
- These technologies vary in their scope: some are tailored to specific application domains like website design and implementation; others concentrate on technology domains such as object-oriented systems or aspect-oriented programming; while some are more overarching, like operating systems such as Linux.

1.1 The Nature of Software

- Software is both a product and a vehicle that delivers a product.

- Today, software serves a dual purpose. It functions both as a product in itself and as the means to deliver other products. In its capacity as a product, software harnesses the computing capabilities provided by computer hardware or, more broadly, by interconnected computers accessible through local hardware.

- Software serves as an information processor, handling tasks such as generating, managing, acquiring, modifying, displaying, or transmitting information. This information can range from simple binary data to intricate multimedia presentations compiled from data gathered from various independent sources.

- Furthermore, as the medium for delivering products, software plays a crucial role in controlling computers through operating systems, facilitating information exchange via networks, and enabling the creation and management of additional programs through software tools and environments.

1.1.1 Defining the software

Software is:

(1) Instructions (computer programs) that when executed provide desired features, function, and performance;

(2) Data structures that enable the programs to adequately manipulate information, and

(3) Descriptive information in both hard copy and virtual forms that describes the operation and use of the programs.

How is Software different from Hardware?

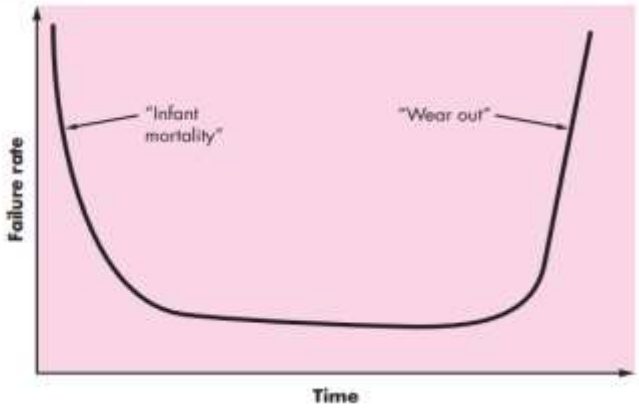
1. Software is developed or engineered; it is not manufactured in the classical sense.

   All though many similarities exist between software and Hardware, Manufacturing phase for hardware can introduce quality problems that doesn't exist in software

2. Software doesn't wear out, but it does deteriorate

   Figure 1.1 depicts failure rate as a function of time for hardware. The relationship, often called the "bathtub curve," indicates that hardware exhibits relatively high failure rates early in its life (these failures are often attributable to design or manufacturing defects); defects are corrected and the failure rate drops to a steady-state level (hopefully, quite low) for some period of time.

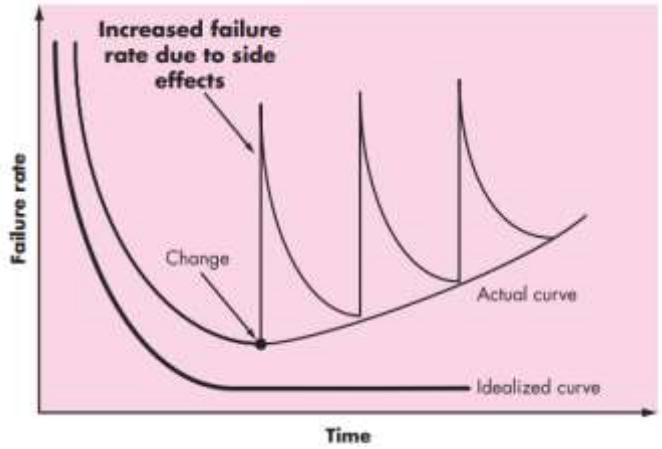
FIGURE 1.1
Failure curve for hardware



   As time passes, however, the failure rate rises again as hardware components suffer from the cumulative effects of dust, vibration, abuse, temperature extremes, and many other environmental maladies. Stated simply, the hardware begins to wear out.

Software is not susceptible to the environmental maladies that cause hardware to wear out. In theory, therefore, the failure rate curve for software should take the form of the "idealized curve" shown in Figure 1.2.

FIGURE 1.2
Failure curves for software



3. Although the industry is moving toward component-based construction, most software continues to be custom built

   As an engineering discipline evolves, a collection of standard design components is created. The reusable components have been created so that the engineer can concentrate on the truly innovative elements of a design, that is, the parts of the design that represent something new. A software component should be designed and implemented so that it can be reused in many different programs.

1.1.2 Software Application Domains

Seven broad categories of computer software present continuing challenges for software engineers:

1. System software: a collection of programs written to service other programs. Some system software (e.g., compilers, editors, and file management utilities) processes complex. Other systems applications (e.g., operating system components, drivers, networking software, telecommunications processors).
2. Application software: Stand-alone programs that solve a specific business need. Applications in this area process business or technical data in a way that facilitates business operations or management/technical decision making.
3. Engineering/scientific software: has been characterized by "number crunching" algorithms. Applications range from astronomy to

volcanology, from automotive stress analysis to space shuttle orbital dynamics, and from molecular biology to automated manufacturing.

4. Embedded software: Resides within a product or system and is used to implement and control features and functions for the end user and for the system itself. (e.g., key pad control for a microwave oven).
5. Product-line software: Designed to provide a specific capability for use by many different customers. Product-line software can focus on a limited and esoteric marketplace (e.g., inventory control products)
6. Web applications: Web-Apps can be little more than a set of linked hypertext files that present information using text and limited graphics.
7. Artificial intelligence software: Makes use of non-numerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis. Applications within this area include robotics, expert systems, pattern recognition (image and voice), artificial neural networks, theorem proving, and game playing.

1.1.3 Legacy Software

In the preceding section, we discussed seven broad application domains encompassing hundreds of thousands of computer programs. Among these, some represent cutting-edge software, recently released for use by individuals, industries, and governmental entities. However, many other programs are considerably older.

These older programs, commonly known as legacy software, have been a focal point of ongoing attention and concern since the 1960s. Legacy software is described as systems developed decades ago and continually modified to adapt to changes in business requirements and computing platforms. The widespread presence of such systems presents challenges for large organizations, which find them expensive to maintain and risky to update.

Expanding on this, many legacy systems still play crucial roles in supporting core business functions and are deemed "indispensable" to business operations. Thus, legacy software is characterized by its longevity and critical importance to business operations.

Legacy systems frequently undergo evolution due to one or more of the following factors:

1. The need to adjust the software to suit new computing environments or technologies.
2. The requirement to incorporate enhancements to accommodate new business needs.
3. The necessity to expand the software to ensure compatibility with other contemporary systems or databases.
4. The need to restructure the software architecture to ensure viability within a networked environment.

1.2 Unique nature of Web-Apps

Characteristics that differentiates Web-Apps from other software:

1. **Network intensiveness:** A Web-App resides on a network and must serve the needs of a diverse community of clients. Eg: Corporate Intranet, Internet.
2. **Concurrency:** A large number of users may access the Web-App at one time.
3. **Unpredictable load:** The number of users of the Web-App may vary by orders of magnitude from day to day.
4. **Performance:** If a Web-App user must wait too long (for access, for server side processing, for client-side formatting and display), he or she may decide to go elsewhere.
5. **Availability:** Although expectation of 100 percent availability is unreasonable, users of popular Web-Apps often demand access on a 24/7/365 basis.
6. **Data driven:** The primary function of many Web-Apps is to use hypermedia to present text, graphics, audio, and video content to the end user.
7. **Content sensitive:** The quality and aesthetic nature of content remains an important determinant of the quality of a Web-App.
8. **Continuous evolution:** Unlike conventional application software that evolves over a series of planned, chronologically spaced releases, Web applications evolve continuously.

9. **Immediacy:** The compelling need to get software to market quickly—is a characteristic of many application domains, Web-Apps often exhibit a time-to-market that can be a matter of a few days or weeks.
10. **Security:** In order to protect sensitive content and provide secure modes of data transmission, strong security measures must be implemented throughout the infrastructure that supports a Web-App and within the application itself.
11. **Aesthetics:** An undeniable part of the appeal of a Web-App is its look and feel. When an application has been designed to market or sell products or ideas, aesthetics may have as much to do with success as technical design.

1.3 Software Engineering

In order to build software that is ready to meet the challenges of the twenty-first century, you must recognize a few simple realities:

1. **A Concerted effort should be made to understand the problem before a software solution is developed:** When a new application or embedded system is to be built, many voices must be heard. And it sometimes seems that each of them has a slightly different idea of what software features and functions should be delivered.
2. **Design becomes a pivotal activity:** Sophisticated software that was once implemented in a predictable, self-contained, computing environment is now embedded inside everything from consumer electronics to medical devices to weapons systems. The complexity of these new computer-based systems and products demands careful attention to the interactions of all system elements.
3. **Software should exhibit high quality:** Individuals, businesses, and governments increasingly rely on software for strategic and tactical decision making as well as day-to-day operations and control. If the software fails, people and major enterprises can experience anything from minor inconvenience to catastrophic failures.
4. **Software should be maintainable:** As the perceived value of a specific application grows, the likelihood is that its user base and longevity will also grow. As its user base and time-in-use increase, demands for adaptation and enhancement will also grow.

Conclusion: Software in all of its forms and across all of its application domains should be engineered.

DEFINITIONS OF SOFTWARE ENGINEERING

1. By Fritz Bauer

Software engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.

2. IEEE [IEE93a]

The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.

Software engineering encompasses a process, methods for managing and engineering software, and tools.

SOFTWARE ENGINEERING AS A LAYERED TECHNOLOGY

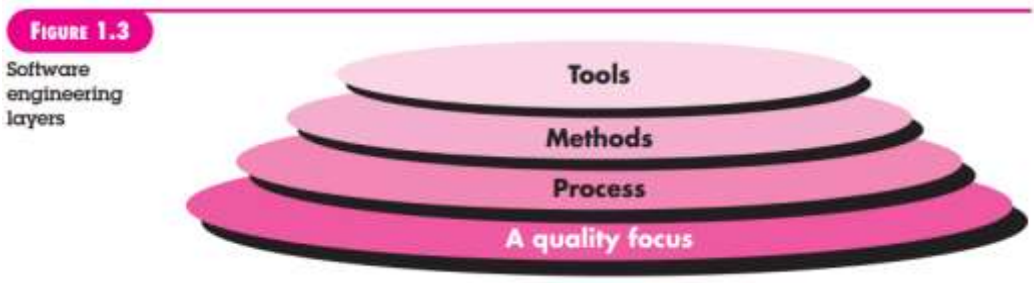


FIGURE 1.3 Software engineering layers

Referring to Figure 1.3, any engineering approach (including software engineering) must rest on an organizational commitment to quality. The bedrock that supports software engineering is a ***quality focus.***

The foundation for software engineering is the **process layer**. The software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software. Process defines a framework that must be established for effective delivery of software engineering technology.

Software engineering ***methods*** provide the technical how-to's for building software. Methods encompass a broad array of tasks that include communication, requirements analysis, design modelling, program construction, testing, and support.

Software engineering **tools** provide automated or semi-automated support for the process and the methods.

1.4 The Software Process

Elements of a software process

1. A process is a collection of activities, actions, and tasks that are performed when some work product is to be created.
2. An activity strives to achieve a broad objective (e.g., communication with stakeholders) and is applied regardless of the application domain, size of the project, complexity of the effort, or degree of rigor with which software engineering is to be applied.
3. An action (e.g., architectural design) encompasses a set of tasks that produce a major work product (e.g., an architectural design model).
4. A task focuses on a small, but well-defined objective (e.g., conducting a unit test) that produces a tangible outcome

A process framework establishes the foundation for a complete software engineering process by identifying a small number of framework activities that are applicable to all software projects, regardless of their size or complexity.

Five generic process framework activities:

1. **Communication:** Before any technical work can commence, it is critically important to communicate and collaborate with the customer and other stakeholders. The intent is to understand stakeholders' objectives for the project and to gather requirements that help define software features and functions.
2. **Planning:** The planning activity creates a "map" that helps guide the team as it makes the journey. The map is called a software project plan defines the software engineering work by describing the technical tasks to be conducted, the risks that are likely, the resources that will be required, the work products to be produced, and a work schedule.
3. **Modelling:** To better understand the problem and how it's going to be solved, a software engineer creates models to better understand software requirements and the design that will achieve those requirements.

4. **Construction:** This activity combines code generation (either manual or automated) and the testing that is required to uncover errors in the code.
5. **Deployment:** The software (can be an increment) is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation.

Typical umbrella activities include:

Umbrella activities occur throughout the software process and focus primarily on project management, tracking, and control.

1. **Software project tracking and control**—allows the software team to assess progress against the project plan and take any necessary action to maintain the schedule.
2. **Risk management**—assesses risks that may affect the outcome of the project or the quality of the product.
3. **Software quality assurance**—defines and conducts the activities required to ensure software quality.
4. **Technical reviews**—assesses software engineering work products in an effort to uncover and remove errors before they are propagated to the next activity.
5. **Measurement**—defines and collects process, project, and product measures that assist the team in delivering software that meets stakeholders' needs.
6. **Software configuration management**—manages the effects of change throughout the software process.
7. **Reusability management**—defines criteria for work product reuse and establishes mechanisms to achieve reusable components.
8. **Work product preparation and production**—encompasses the activities required to create work products such as models, documents, logs, forms, and lists.

1.5 Software Engineering Practice

1.5.1 The Essence of practice:

1. Understand the problem (communication and analysis).
2. Plan a solution (modelling and software design).
3. Carry out the plan (code generation).
4. Examine the result for accuracy (testing and quality assurance).

Answering few questions below gives a clear picture of essence of practice.

Understand the problem:

- Who has a stake in the solution to the problem? That is, who are the stakeholders?
- What are the unknowns? What data, functions, and features are required to properly solve the problem?
- Can the problem be compartmentalized? Is it possible to represent smaller problems that may be easier to understand?
- Can the problem be represented graphically? Can an analysis model be created?

Plan the solution:

- Have you seen similar problems before? Are there patterns that are recognizable in a potential solution? Is there existing software that implements the data, functions, and features that are required?
- Has a similar problem been solved? If so, are elements of the solution reusable?
- Can sub-problems be defined? If so, are solutions readily apparent for the sub-problems?
- Can you represent a solution in a manner that leads to effective implementation? Can a design model be created?

Carry out the plan:

- Does the solution conform to the plan? Is source code traceable to the design model?
- Is each component part of the solution provably correct? Have the design and code been reviewed, or better, have correctness proofs been applied to the algorithm?

Examine the result:

- Is it possible to test each component part of the solution? Has a reasonable testing strategy been implemented?
- Does the solution produce results that conform to the data, functions, and features that are required? Has the software been validated against all stakeholder requirements?

1.5.2 The General Principles

David Hooker has put forward seven principles that centre on the overall practice of software engineering. These principles are outlined in the following paragraphs.

1. The First Principle: The Reason It All Exists

The primary purpose of a software system is to deliver value to its users. All decisions regarding the system should be guided by this principle. Whether it involves specifying system requirements, defining system functionality, or selecting hardware platforms and development processes, the focus should always be on delivering value to the users.

2. The Second Principle: KISS (Keep It Simple, Stupid!)

Software design is a meticulous process that requires consideration of numerous factors. It's essential to strive for simplicity in design, but not at the cost of oversimplification. This approach ensures the creation of a system that is both easily comprehensible and maintainable.

3. The Third Principle: Maintain the Vision

A clear vision is crucial for the success of a software project. Without it, the project is prone to internal conflicts and uncertainties, leading to inefficiencies and potential failure.

4. The Fourth Principle: What You Produce, Others Will Consume

Rarely does the development and utilization of an industrial-strength software system occur in isolation. There is almost always involvement from others who will either use, maintain, document, or rely on understanding the system. Therefore, it's imperative to always specify, design, and implement with the awareness that someone else will need to comprehend the work being done.

5. The Fifth Principle: Be Open to the Future

Long-lasting software systems hold greater value. While modern computing environments see rapid changes, industrial-grade software must endure much longer. Successful systems adapt to evolving requirements from the start, avoiding design constraints and preparing for various scenarios. By solving broader problems, they enable potential reuse of entire systems.

6. The Sixth Principle: Plan Ahead for Reuse

Achieving significant reuse is a challenging goal in software development, but it can save considerable time and effort. While object-oriented technologies offer the promise of code and design reuse, realizing this benefit requires careful planning and consideration.

7. The Seventh principle: Think!

The importance of careful consideration is often underestimated. Prioritizing clear and thorough thinking before taking action typically leads to superior outcomes. Engaging in thoughtful reflection increases the likelihood of executing tasks correctly and acquiring knowledge for future endeavours. Even if an action results in a mistake despite prior thought, it becomes a valuable learning experience. Additionally, thinking helps in recognizing gaps in knowledge, prompting further research to find solutions.

Chapter 2

Process Models

2.1 Generic Process Models

According to Figure 2.1, the software process is depicted schematically. In this representation, each framework activity is filled with a collection of software engineering actions. These actions are delineated by a task set, which specifies the work tasks to be undertaken, the resulting work products, the necessary quality assurance checkpoints, and the milestones that denote progress.

A general process framework for software engineering outlines five core framework activities: communication, planning, modelling, construction, and deployment. Additionally, a series of overarching umbrella activities, including project tracking and control, risk management, quality assurance, configuration management, technical reviews, and others, are incorporated across the entire process.

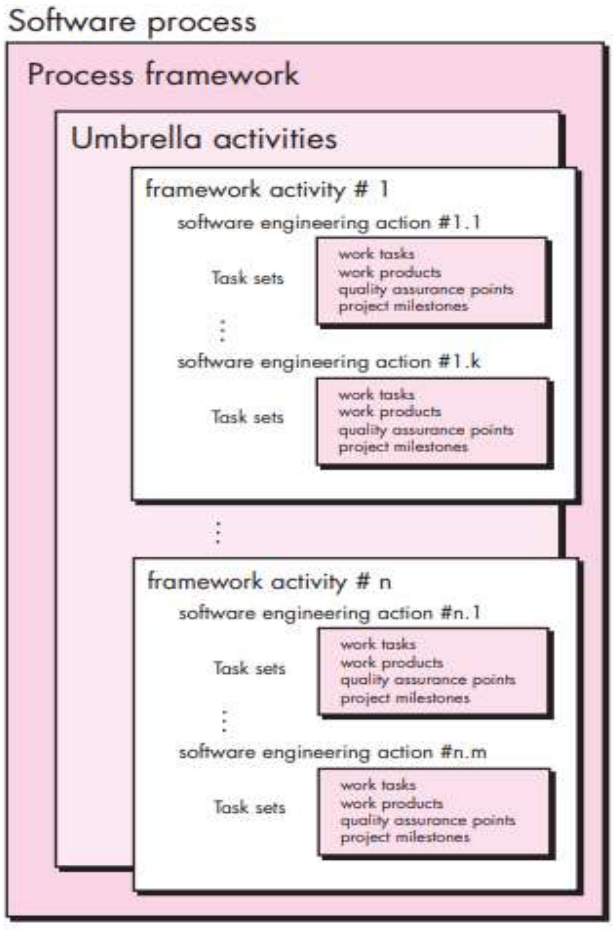
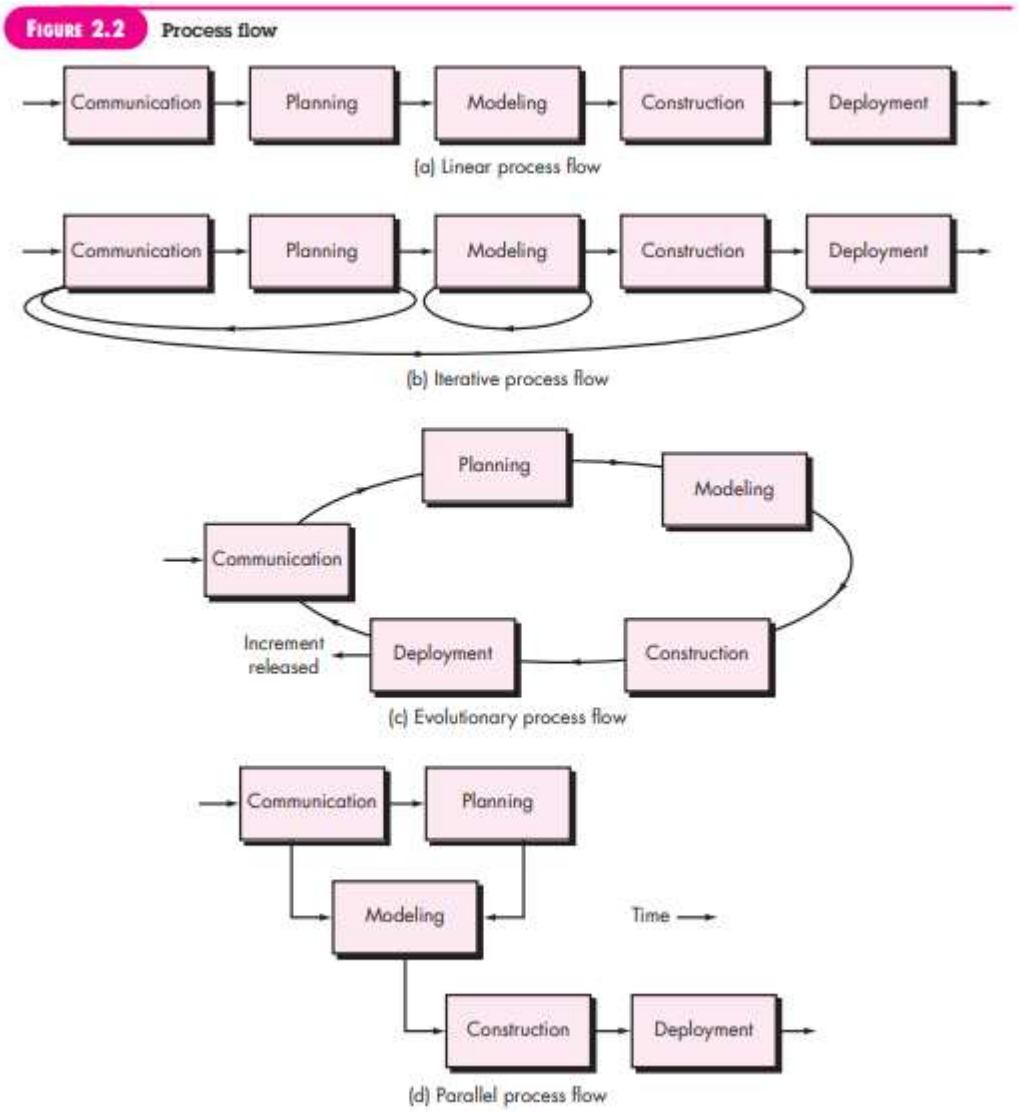


FIGURE 2.1 A software process framework

- In a linear process flow (Figure 2.2a), the five framework activities are executed sequentially, starting with communication and ending with deployment.
- An iterative process flow (Figure 2.2b) involves repeating one or more activities before advancing to the next stage.
- An evolutionary process flow (Figure 2.2c) follows a circular pattern where each cycle through the five activities results in a more refined version of the software.
- In a parallel process flow (Figure 2.2d), activities are executed simultaneously, such as modelling and construction for different aspects of the software.



2.1.1 Defining a Framework Activity

For a small software project with straightforward requirements from a single remote stakeholder, communication may involve just a phone call. The task set for this action includes:

1. Contacting the stakeholder via telephone.
2. Discussing requirements and noting them down.

As the complexity of the project increases, with multiple stakeholders having diverse and sometimes conflicting requirements, the communication activity would expand. It might include additional tasks such as:

3. Compiling notes into a concise written statement of requirements.
4. Sending the statement to stakeholders for review and approval.

2.1.2 Identifying a Task Set

In Figure 2.1, each software engineering action, such as elicitation linked to the communication activity, can be depicted through different task sets. These sets include various software engineering tasks, related work products, quality assurance points, and project milestones. It's crucial to select the most suitable task set according to the project's needs and team characteristics. This suggests that software engineering actions can be customized to fit the project's specific requirements and the team's attributes.

2.1.3 Process Patterns

A process pattern describes a process-related problem that is encountered during software engineering work, identifies the environment in which the problem has been encountered, and suggests one or more proven solutions to the problem.

Stated in more general terms, a process pattern provides you with a template- a consistent method for describing problem solutions within the context of the software process. By combining patterns, a software team can solve problems and construct a process that best meets the needs of a project.

Template for describing a process pattern:

- Pattern Name: The pattern is given a meaningful name describing it within the context of the software process (e.g., **TechnicalReviews**).

- Forces: The environment in which the pattern is encountered and the issues that make the problem visible and may affect its solution.
- Type: The pattern type is specified. There are 3 types of patterns:
  1. **Stage pattern**—defines a problem associated with a framework activity for the process. . An example of a stage pattern might be **EstablishingCommunication**. This pattern would incorporate the task pattern **RequirementsGathering** and others.
  2. **Task pattern**—defines a problem associated with a software engineering action or work task and relevant to successful software engineering practice (e.g., **RequirementsGathering**)
  3. **Phase pattern**—define the sequence of framework activities that occurs within the process, even when the overall flow of activities is iterative in nature. An example of a phase pattern might be **SpiralModel or Prototyping.**

Initial context: Describes the conditions under which the pattern applies. Prior to the initiation of the pattern:

(1) What organizational or team-related activities have already occurred?
(2) What is the entry state for the process?
(3) What software engineering information or project information already exists?
For example, the Planning pattern (a stage pattern) requires that

(1) Customers and software engineers have established a collaborative communication.
(2) Successful completion of a number of task patterns for the Communication pattern has occurred; and
(3) The project scope, basic business requirements, and project constraints are known.

Problem: The specific problem to be solved by the pattern.

Solution: Describes how to implement the pattern successfully. This describes how the initial state of the process (that exists before the pattern is implemented) is modified as a consequence of the initiation of the pattern. It also describes how software engineering information or project information that is available before the initiation of the pattern is transformed as a consequence of the successful execution of the pattern.

Resulting Context: Describes the conditions that will result once the pattern has been successfully implemented. Upon completion of the pattern:

(1) What organizational or team-related activities must have occurred?
(2) What is the exit state for the process?
(3) What software engineering information or project information has been developed?

Related Patterns: Provide a list of all process patterns that are directly related to this one. This may be represented as a hierarchy or in some other diagrammatic form. For example, the stage pattern Communication encompasses the task patterns: **ProjectTeam, CollaborativeGuidelines, ScopeIsolation, RequirementsGathering, ConstraintDescription, and ScenarioCreation.**

Known Uses and Examples: Indicates the specific instances in which the pattern is applicable. For example, Communication is mandatory at the beginning of every software project, is recommended throughout the software project, and is mandatory once the deployment activity is under way.

2.2 Process assessment and improvement

Assessment attempts to understand the current state of the software process with the intent of improving it.

Formal techniques available for assessing the software process:

1. **Standard CMMI Assessment Method for Process Improvement (SCAMPI)—**provides a five-step process assessment model that incorporates five phases: initiating, diagnosing, establishing, acting, and learning.
2. **CMM-Based Appraisal for Internal Process Improvement (CBA IPI) —** provides a diagnostic technique for assessing the relative maturity of a software organization.
3. **SPICE (ISO/IEC15504)—**a standard that defines a set of requirements for software process assessment. The intent of the standard is to assist organizations in developing an objective evaluation.
4. **ISO 9001:2000 for Software—**a generic standard that applies to any organization that wants to improve the overall quality of the products, systems, or services that it provides.

2.3 Prescriptive Process Models

- It is called as Prescriptive because they prescribe a set of process elements—framework activities, software engineering actions, tasks, work products, quality assurance, and change control mechanisms for each project. Each process model also prescribes a process flow (also called a work flow)—that is, the manner in which the process elements are interrelated to one another.

- Prescriptive process models define a prescribed set of process elements and a predictable process work flow.

2.3.1 The Waterfall Model

- The waterfall model is called as classic life cycle, suggests a systematic, sequential approach to software development that begins with customer specification of requirements and progresses through planning, modelling, construction, and deployment, culminating in ongoing support of the completed software (Figure 2.3)

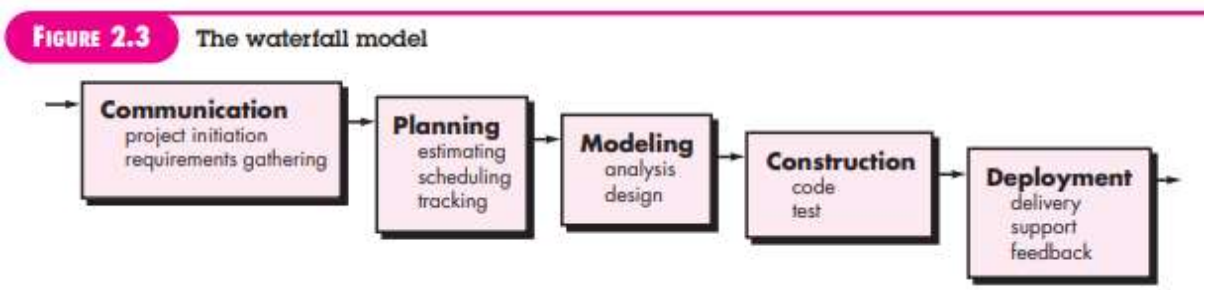
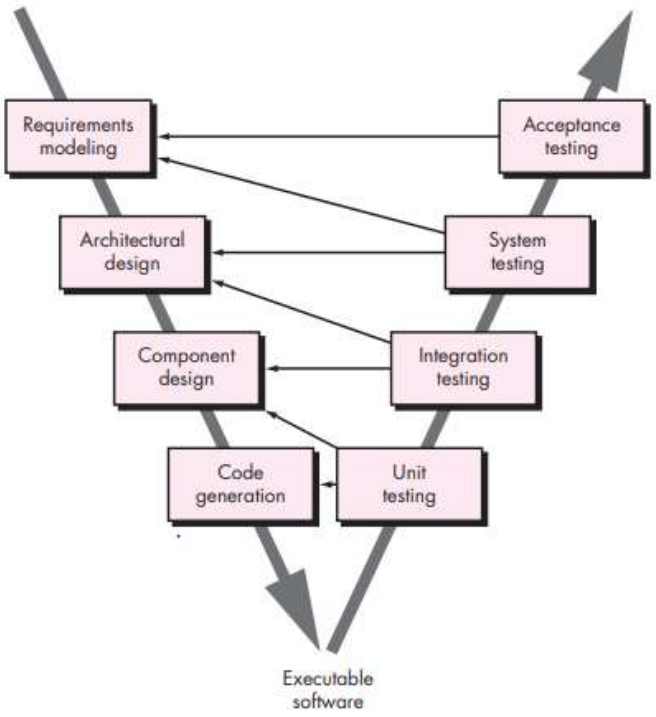


FIGURE 2.3 The waterfall model

- A variation in the representation of the waterfall model is called the V-model.
- The V-model illustrates how verification and validation actions are associated with earlier engineering actions. Figure 2.4 depicts the V-model describing the relationship of quality assurance actions to the actions associated with communication, modelling, and early construction activities.
- As a software team moves down the left side of the V, basic problem requirements are refined into progressively more detailed and technical representations of the problem and its solution.
- Once code has been generated, the team moves up the right side of the V, essentially performing a series of tests (quality assurance actions) that validate each of the models created as the team moved down the left side.
- The V-model provides a way of visualizing how verification and validation actions are applied to earlier engineering work.

FIGURE 2.4
The V-model



Reasons for the failure of waterfall model:

1. Real projects rarely follow the sequential flow that the model proposes.
2. It is often difficult for the customer to state all requirements explicitly.
3. The customer must have patience.
4. It is found that the linear nature of the classic life cycle leads to "blocking states" in which some project team members must wait for other members of the team to complete dependent tasks. Time spent may exceed the time taken for production sometimes.

2.3.2 Incremental Process Models

- The incremental model delivers a series of releases, called increments that provide progressively more functionality for the customer as each increment is delivered.
- The incremental model combines elements of linear and parallel process flows.
- For example, word-processing software developed using the incremental paradigm might deliver basic file management, editing, and document production functions in the first increment; more sophisticated editing and document production capabilities in the second increment; spelling and grammar checking in the third increment; and advanced page layout capability in the fourth increment.

- When an incremental model is used, the first increment is often a core product. That is, basic requirements are addressed but many supplementary features remain undelivered.
- The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality. This process is repeated following the delivery of each increment, until the complete product is produced.
- The incremental process model focuses on the delivery of an operational product with each increment.

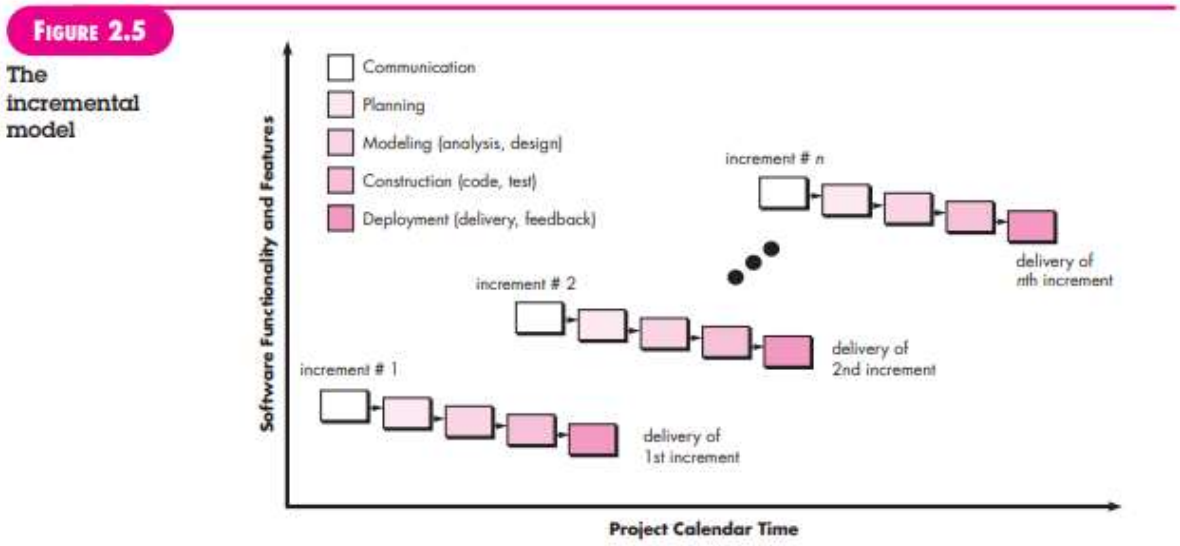


FIGURE 2.5 The incremental model

2.3.3 Evolutionary Process Models

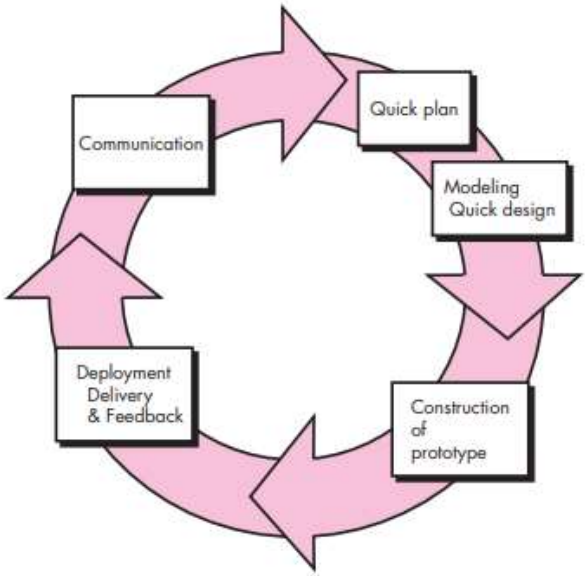
- Evolutionary process models produce an increasingly more complete version of the software with each iteration.
- Business and product requirements often change as development proceeds; tight market deadlines make completion of a comprehensive software product impossible.
- Evolutionary models are iterative. They are characterized in a manner that enables you to develop increasingly more complete versions of the software.

Prototyping:

- When your customer has a legitimate need, but is clueless about the details, develop a prototype as a first step.
- A customer defines a set of general objectives for software, but does not identify detailed requirements for functions and features.

- A prototyping iteration is planned quickly, and modelling (in the form of a "quick design") occurs. A quick design focuses on a representation of those aspects of the software that will be visible to end users (e.g., human interface layout or output display formats).
- The quick design leads to the construction of a prototype. The prototype is deployed and evaluated by stakeholders, who provide feedback that is used to further refine requirements. Iteration occurs as the prototype is tuned to satisfy the needs of various stakeholders.
- Ideally, the prototype serves as a mechanism for identifying software requirements.

FIGURE 2.6
The prototyping paradigm



Problems associated with Prototyping:

1. Stakeholders see what appears to be a working version of the software not considering the over-all software quality and long term maintainability.
2. Implementation compromises are made in order to get a prototype working quickly; inefficient algorithm might be used; inappropriate OS or Programming language might be used.

If all the stakeholders agree that the prototype is built to serve as a mechanism for defining requirements, then prototyping can be an effective paradigm for software engineering.

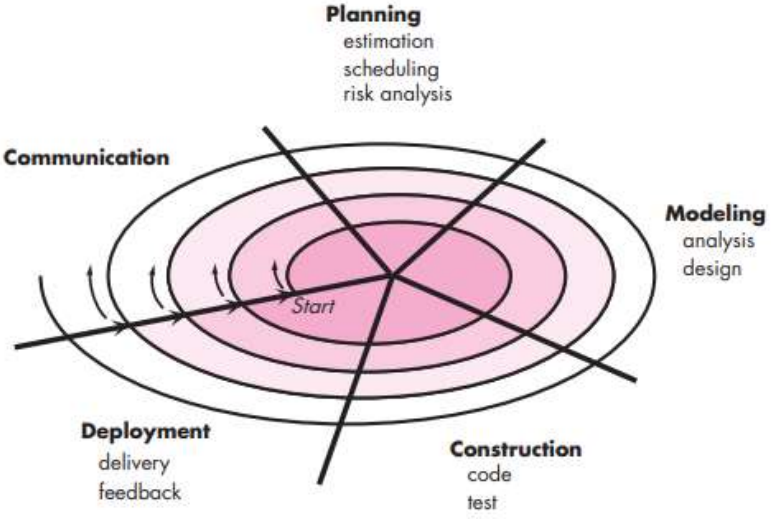
The Spiral Model:

- The spiral model is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model.
- The spiral model can be adapted to apply throughout the entire life cycle of an application, from concept development to maintenance.
- As this evolutionary process begins, the software team performs activities that are implied by a circuit around the spiral in a clockwise direction, beginning at the centre.
- Risk is analysed as each revolution is made.
- Project milestones are attained along the path of the spiral after each pass.
- The first circuit around the spiral might result in the development of a product specification; subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software.
- Each pass through the planning region results in adjustments to the project plan. Cost and schedule are adjusted based on feedback derived from the customer after delivery.
- The spiral model is a realistic approach to the development of large-scale systems and software.

Features:

1. Risk-driven process model generator
2. It maintains the systematic stepwise approach but incorporates it into an iterative framework.
3. Guides multi-stakeholder
4. Concurrent in nature
5. cyclic approach
6. Incrementally growing
7. Ensures to meet the project milestones

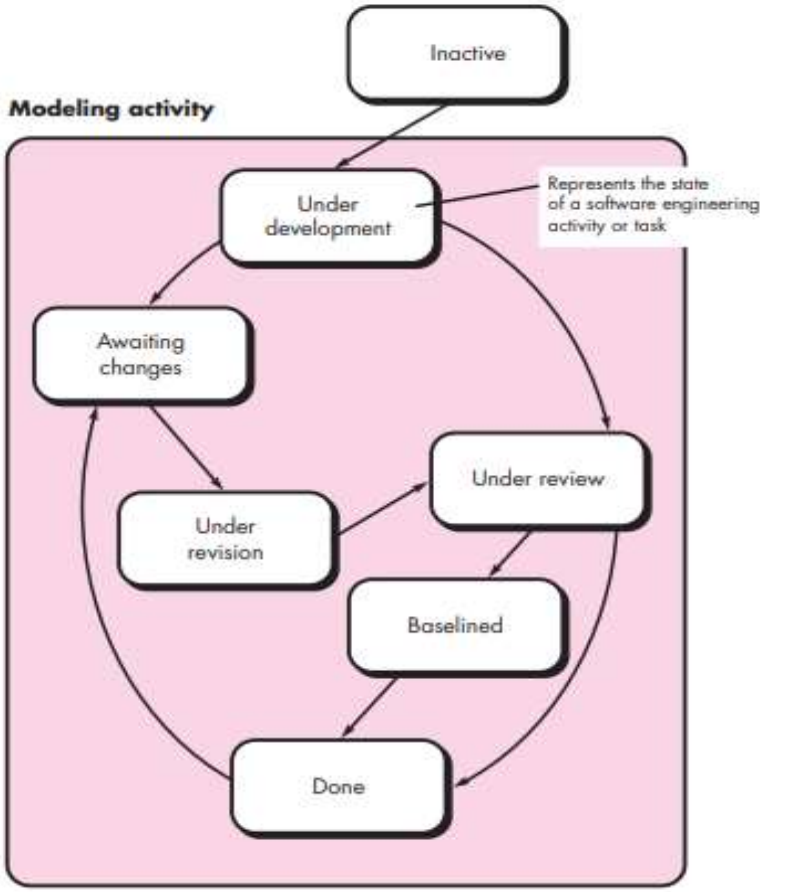
FIGURE 2.7
A typical spiral model



2.3.4 Concurrent Models

- The concurrent model is often more appropriate for product engineering projects where different engineering teams are involved. Figure 2.8 provides a schematic representation of one software engineering activity within the modelling activity using a concurrent modelling approach.

FIGURE 2.8
One element of the concurrent process model



- Modelling activity may be in any one of the states noted at any given time. Similarly, other activities, actions, or tasks (e.g., communication or construction) can be represented in an analogous manner. All software engineering activities exist concurrently but reside in different states.
- For example, assuming that project the communication activity has completed its first iteration and currently in the awaiting changes state. The modelling activity (which was in the inactive state while initial makes a transition into the under development state. If, however, the customer indicates that changes in requirements must be made, the modelling activity moves from the under development state into the awaiting changes state.
- A series of events is going to trigger transitions from state to state for each of the software engineering activities, actions, or tasks. Concurrent modelling is applicable to all types of software development and provides an accurate picture of the current state of a project.

2.3.5 A Final Word on Evolutionary Processes

Weaknesses of Evolutionary Process Model:

1. Poses a problem to project planning because of the uncertain number of cycles required to construct the product.
2. Evolutionary software processes do not establish the maximum speed of the evolution.
3. Software processes should be focused on flexibility and extensibility rather than on high quality.

2.4 Specialized process models

2.4.1 Component-Based Development

The component-based development model incorporates many of the characteristics of the spiral model. It is evolutionary in nature, demanding an iterative approach to the creation of software.

Commercial off-the-shelf (COTS) software components are pre-developed software products created by third-party vendors. These components are designed to offer specific functionalities and come with clearly defined interfaces, making them easy to integrate into larger software systems being developed.

The component-based development model includes these steps:

1. Investigate and assess available component-based products for the specific application domain.

2. Consider potential challenges related to component integration.

3. Develop a software architecture that includes the identified components.

4. Integrate the components into the established architecture.

5. Conduct extensive testing to confirm proper functionality.

The component-based development model encourages software reuse, which offers software engineers various quantifiable benefits.

2.4.2 The Formal Methods Model

- The formal methods model includes a series of steps that produce a mathematical specification of computer software. These methods use rigorous mathematical notation to specify, develop, and verify computer-based systems.
- During the development process, formal methods provide a way to handle numerous challenges that are challenging to address using alternative software engineering methods. They aid in identifying and resolving problems like ambiguity, incompleteness, and inconsistency with greater efficiency.
- When employed in the design phase, formal methods act as a foundation for program verification, allowing the identification and correction of errors that might otherwise remain unnoticed.

Problems to be addressed:

- Creating formal models is presently a time-consuming and costly endeavour.
- Due to the limited number of software developers equipped with the requisite expertise in applying formal methods, extensive training is necessary.
- Communicating the models to technically inexperienced clients poses challenges.

2.4.3 Aspect-Oriented Software Development

AOSD defines "aspects" as representations of customer concerns that span across various system functions, features, and information.

Aspect-oriented software development (AOSD), also known as aspect-oriented programming (AOP), is a modern software engineering paradigm that offers a structured approach and methodology for delineating, specifying, designing, and building aspects.