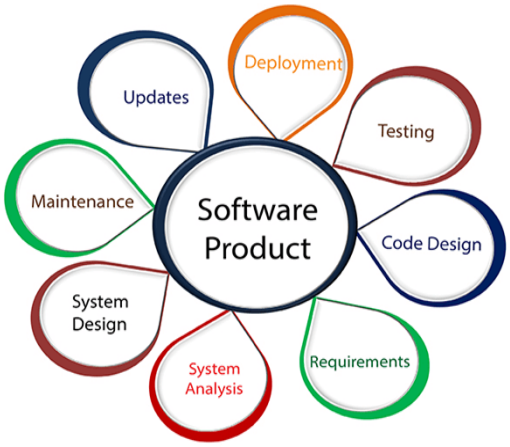
➢ What is Software Engineering?



The term software engineering is the product of two words, software, and engineering.

The software is a collection of integrated programs.

Software subsists of carefully-organized instructions and code written by developers on any of various particular computer languages.

Computer programs and related documentation such as requirements, design models and user manuals Engineering is the application of scientific and practical knowledge to invent, design, build, maintain, and improve frameworks, processes, etc.

Software Engineering is an engineering branch related to the evolution of software product using well-defined scientific principles, techniques, and procedures. The result of software engineering is an effective and reliable software product.

Why is Software Engineering required?

Software Engineering is required due to the following reasons:

- To manage Large software
- For more Scalability
- Cost Management
- To manage the dynamic nature of software
- For better quality Management

Need of Software Engineering

The necessity of software engineering appears because of a higher rate of progress in user requirements and the environment on which the program is working.

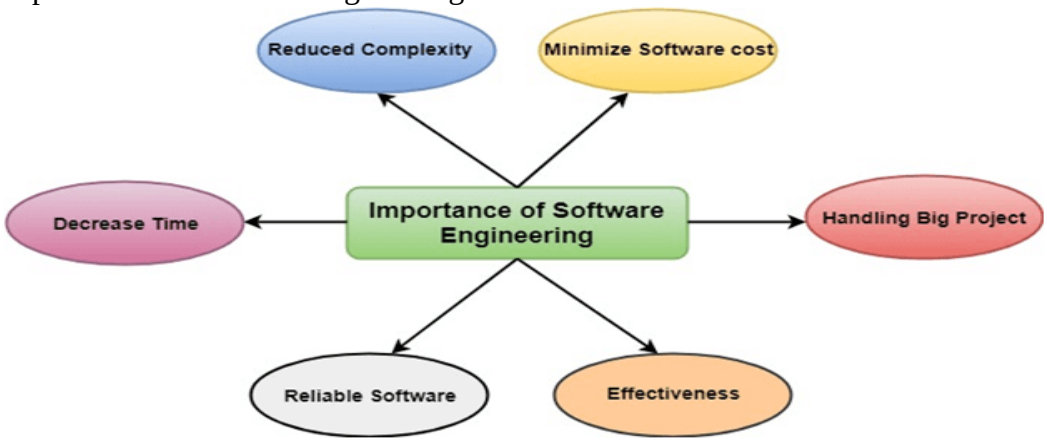
- **Huge Programming:** It is simpler to manufacture a wall than to a house or building, similarly, as the measure of programming become extensive engineering has to step to give it a scientific process.
- **Adaptability:** If the software procedure were not based on scientific and engineering ideas, it would be simpler to re-create new software than to scale an existing one.
- **Cost:** As the hardware industry has demonstrated its skills and huge manufacturing has let down the cost of computer and electronic hardware. But the cost of programming remains high if the proper process is not adapted.
- **Dynamic Nature:** The continually growing and adapting nature of programming hugely depends upon the environment in which the client works. If the quality of the software is continually changing, new upgrades need to be done in the existing one.
- **Quality Management:** Better procedure of software development provides a better and quality software product.

Characteristics of a good software engineer

Exposure to systematic methods, i.e., familiarity with software engineering principles.

- Good technical knowledge of the project range (Domain knowledge).
- Good programming abilities.
- Good communication skills. These skills comprise of oral, written, and interpersonal skills.
- High motivation.
- Sound knowledge of fundamentals of computer science.
- Intelligence.
- Ability to work in a team

Importance of Software Engineering



The importance of Software engineering is as follows:

1. **Reduces complexity:** Big software is always complicated and challenging to progress. Software engineering has a great solution to reduce the complication of any project. Software engineering divides big problems into various small issues. And then start solving each small issue one by one. All these small problems are solved independently to each other.
2. **To minimize software cost:** Software needs a lot of hardwork and software engineers are highly paid experts. A lot of manpower is required to develop software with a large number of codes. But in software engineering, programmers project everything and decrease all those things that are not needed. In turn, the cost for software productions becomes less as compared to any software that does not use software engineering method.
3. **To decrease time:** Anything that is not made according to the project always wastes time. And if you are making great software, then you may need to run many codes to get the definitive running code. This is a very time-consuming procedure, and if it is not well handled, then this can take a lot of time. So if you are making your software according to the software engineering method, then it will decrease a lot of time.
4. **Handling big projects:** Big projects are not done in a couple of days, and they need lots of patience, planning, and management. And to invest six and seven months of any company, it requires heaps of planning, direction, testing, and maintenance. No one can say that he has given four months of a company to the task, and the project is still in its first stage. Because the company has provided many resources to the plan and it should be completed. So to handle a big project without any problem, the company has to go for a software engineering method.
5. **Reliable software:** Software should be secure, means if you have delivered the software, then it should work for at least its given time or subscription. And if any bugs come in the software, the company is responsible for solving all these bugs. Because in software engineering, testing and maintenance are given, so there is no worry of its reliability.
6. **Effectiveness:** Effectiveness comes if anything has made according to the standards. Software standards are the big target of companies to make it more effective. So Software becomes more effective in the act with the help of software engineering.

Software myths are misleading attitudes that have caused serious problems for managers and technical people alike.

➢ Software Myths

Software myths propagate misinformation and confusion. There are three kinds of software myths:

1) Management myths: Managers with software responsibility are often under pressure to maintain budgets, keep schedules from slipping, and improve quality. Following are the Management Myths:

• **Myth:** We already have a book that's full of standards and procedures for building software; won't that provide my people with everything they need to know?

Reality: The book of standards may very well exist, but isn't used. Most software practitioners aren't aware of its existence. Also, it doesn't reflect modern software engineering practices and is also complete.

- **Myth**: My people have state-of-the-art software development tools; after all, we buy them the newest computers.

Reality: It takes much more than the latest model mainframe, workstation, or PC to do high-quality software development. Computer-aided software engineering (CASE) tools are more important than hardware for achieving good quality and productivity, yet the majority of software developers still do not use them effectively.

- **Myth:** If we get behind schedule, we can add more programmers and catch up (sometimes called the Mongolian horde concept).

Reality: Software development is not a mechanistic process like manufacturing. As new people are added, people who were working must spend time educating the newcomers, thereby reducing the amount of time spent on productive development effort. People can be added but only in a planned and well-coordinated manner.

- **Myth:** If I decide to outsource the software project to a third party, I can just relax and let that firm build it.

Reality: If an organization does not understand how to manage and control software projects internally, it will invariably struggle when it outsources software projects.

2) Customer myths: Customer myths lead to false expectations (by the customer) and ultimately, dissatisfaction with the developer. Following are the customer myths:

- **Myth**: A general statement of objectives is sufficient to begin writing programs-we can fill in the details later.

Reality: A poor up-front definition is the major cause of failed software efforts. A formal and detailed description of the functions, behavior, performance, interfaces, design constraints, and validation criteria is essential.

- **Myth**: Project requirements continually change, but change can be easily accommodated because software is flexible.

Reality: It is true that software requirements change, but the impact of change varies with the time at which it is introduced. When changes are requested during software design, the cost impact grows rapidly. Resources have been committed and a design framework has been established. Change can cause heavy additional costs. Change, when requested after software is in production, can be much more expensive than the same change requested earlier

3) Practitioner's myths: Practitioners have following myths

Myth: Once we write the program and get it to work, our job is done.

Reality: Industry data indicate that between 60 and 80 percent of all effort expended on software will be expended after it is delivered to the customer for the first time.

Myth: Until I get the program "running" I have no way of assessing its quality.

Reality: One of the most effective software quality assurance mechanisms can be applied from the inception of a project—the formal technical review.

Myth: The only deliverable work product for a successful project is the working program.

Reality: A working program is only one part of a software configuration that includes many element. Documentation provides a foundation for successful engineering and more importantly, guidance for software support.

Myth: Software engineering will make us creates voluminous and unnecessary documentation and will invariably slow us down.

Reality: Software engineering is not about creating documents. It is about creating quality. Better quality leads to reduced rework. And reduced reworks results in faster delivery times.

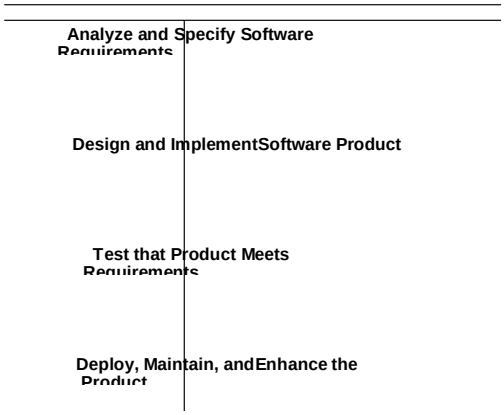
➢ Software Engineering Processes

In order for software to be consistently well engineered, its development must be conducted in an orderly

process. It is sometimes possible for a small software product to be developed without a well-defined process. However, for a software project of any substantial size, involving more than a few people, a good process is essential. The process can be viewed as a road map by which the project participants understand where they are going and how they are going to get there.There is general agreement among software engineers on the major steps of a software process. Figure 1 is a graphical depiction of these steps. As discussed in Chapter 1, the first three steps in the process dia- gram coincide with the basic steps of the problem solving process, as shown in Table 4. The fourth step in the process is the post-development phase, where the product is deployed to its users, maintained as necessary, and enhanced to meet evolving requirements.

The first two steps of the process are often referred to, respectively, as the "what and how" of software development. The "**Analyze and Specify**" step defines *what* the problem is to be solved; the "**Design and Implement**" step entails *how* the problem is solved.

Figure 1: Diagram of general software process steps.



Problem-Solving Phase	Software Process Step
Define the Problem	Analyze and Specify Software Requirements
Solve the Problem	Design and Implement Software Product
Verify the Solution	Test that Product Meets Requirements

Table 4: Correspondence between problem-solving and software processes.

While these steps are common in most definitions of software process, there are wide variations in how process details are defined. The variations stem from the kind of software being developed and the people doing the development. For example, the process for developing a well-understood business application with a highly experienced team can be quite different from the process of developing an experimental artificial intelligence program with a group of academic researchers.

Among authors who write about software engineering processes, there is a good deal of variation in process details. There is variation in terminology, how processes are structured, and the emphasis placed on different aspects of the process. This chapter will define key process terminology and present a specific process that is generally applicable to a range of end-user software. The chapter will also discuss alternative approaches to defining software engineering processes.

Independent of technical details, there are general quality criteria that apply to any good process. These criteria include the following:

1. The process is suited to the people involved in a project and the type of software being developed.
2. All project participants clearly understand the process, or at minimum the part of the process in which they are directly involved.

3. If possible, the process is defined based on the experience of engineers who have participated in successful projects in the past, in an application domain similar to the project at hand.
4. The process is subject to regular evaluation, so that adjustments can be made as necessary during a project, and so the process can be improved for future projects.

with neat graphs and tables, the software development process is intended to appear quite orderly. In actual practice, the process can get messy. Developing software often involves people of diverse backgrounds, varying skills, and differing viewpoints on the product to be developed. Added to this are the facts that software projects can take a long time to complete and cost a lot of money. Given these facts, software development can be quite challenging, and at times trying for those doing it.

Having a well-defined software process can help participants meet the challenges and minimize the trying times. However, any software process must be conducted by people who are willing and able to work effectively with one another. Effective human communication is absolutely essential to any software development project, whatever specific technical process is employed.

General Concepts of Software Processes

Before defining the process followed in the book, some general process concepts are introduced. These concepts will be useful in understanding the definition, as well as in the discussion of different approaches to defining software processes.

Process Terminology

The following terminology will be used in the presentation and discussion of this chapter:

- ***software process:*** a hierarchical collection of *process steps*; hierarchical means that a process step can in turn have *sub-steps*
- ***process step:*** one of the activities of a software process, for example "**Analyze and Specify Software Requirements**" is the first step in Figure 1 ; for clarity and consistency of definition, process steps are named with verbs or verb phrases
- ***software artifact:*** a software work product produced by a process step; for example, a requirements specification document is an artifact produced by the "**Analyze and Specify**" step; for clarity and consistency, process artifacts are named with nouns or noun phrases
- ***ordered step:*** a process step that is performed in a particular order in relation to other steps; the steps shown in Figure 1 are ordered, as indicated by the arrows in the diagram
- ***pervasive step:*** a process step that is performed continuously or at regularly-scheduled intervals throughout the ordered process; for example, process steps to perform project management tasks are pervasive, since management is a continuous ongoing activity
- **process enactment:** the activity of performing a process; most process steps are enacted by people, but some can be automated and enacted by a software development tool
- **step precondition:** a condition that must be true before a process step is enacted; for example, a pre-condition for the "**Design and Implement**" step could be that the requirements specification is signed off by the customer
- **step post condition*:*** a condition that is true after a process step is enacted; for example, a post condition for the "**Design and Implement**" step is that the implementation is complete and ready to be tested for final delivery.

In addition to these specific terms, there is certain general terminology that is used quite commonly in software engineering textbooks and literature. In particular, the terms "analyze", "specify", "design", and "implement" appear nearly universally. While the use of these terms is widespread, their definitions are not always the same. In this book, these terms are given specific definitions in the context of the process that is defined later in this chapter. This book's definitions here are consistent with mainstream usage, however the reader should be aware that specific definitions of these terms can vary among authors.

Process Structure

There are a variety of ways to depict a process. A typical graphical depiction uses a diagram with boxes and arrows, as shown in Figure 1. In this style of diagram, a process step is shown as a rounded box, and the

order of steps is depicted with the arrowed lines. Process sub-steps are typically shown with a box expansion notation. For example, Figure 2 shows the expansion of the "**Analyze and Specify**" step. The activities of the first sub-step include general aspects of requirements analysis, such as defining the overall problem, identifying personnel, and studying related products. The second sub-step defines functional requirements for the way the software actually works, i.e., what functions it performs. The last sub-step defines non-functional requirements, such as how much the product will cost to develop and how long it will take to develop. This expansion is an over-simplification for now, since there are more than three sub-steps in "**Analyze and Specify**". A complete process expansion is coming up a bit later in this chapter.

- A more compact process notation uses mostly text, with indentation and small icons to depict sub-step expansion. Figure 3 shows a textual version of the general process, with the first step partially expanded, and other steps unexpanded. Right-pointing arrowheads depict an unexpanded process step. Down-

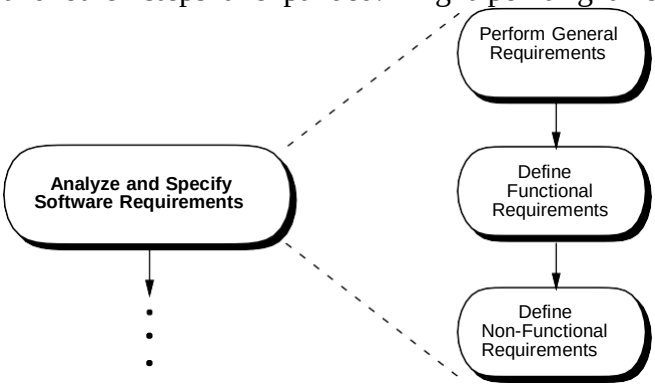


Figure 2: Expansion of the ‘‘Analyze and Specify’’ Step.

Analyze and Specify Software Requirements

Perform General Requirements Analysis

State Problem to be Solved Identify People Involved Analyze Operational Setting Analyze Impacts Identify Positive Impacts Identify Negative Impacts Analyze Related Systems Analyze Feasibility
Define Functional Requirements Define Non-Functional Requirements
Design and Implement Software Product
Test that Product Meets Requirements
Deploy, Maintain, and Enhance the Product

pointing arrow heads depict a process step with its sub-steps expanded immediately below. A round bullet depicts a process step that has no sub-steps.

Depending on the context, one or the other form of process depiction can be useful. When the emphasis is on the flow of the process, the graphical depiction can be most useful. To show complete process details, the textual depiction is generally more appropriate.

An important property of the textual depiction is that it can be considered unordered in terms of process step enactment. In the graphical process depiction, the directed lines connote a specific ordering of steps and sub-steps. The textual version can be considered more abstract, in that the top-to-bottom order of steps does not necessarily depict the specific order in which steps are enacted.

Given its abstractness, the textual depiction of a process can be considered the *canonical form*. Canonical form is a mathematical term meaning the standard or most basic form of something, for which other forms can exist. In the case of a software process, the canonical process form is the one most typically followed. The process can vary from its canonical form in terms of the order in which the steps are followed, and the number of times steps may be repeated.

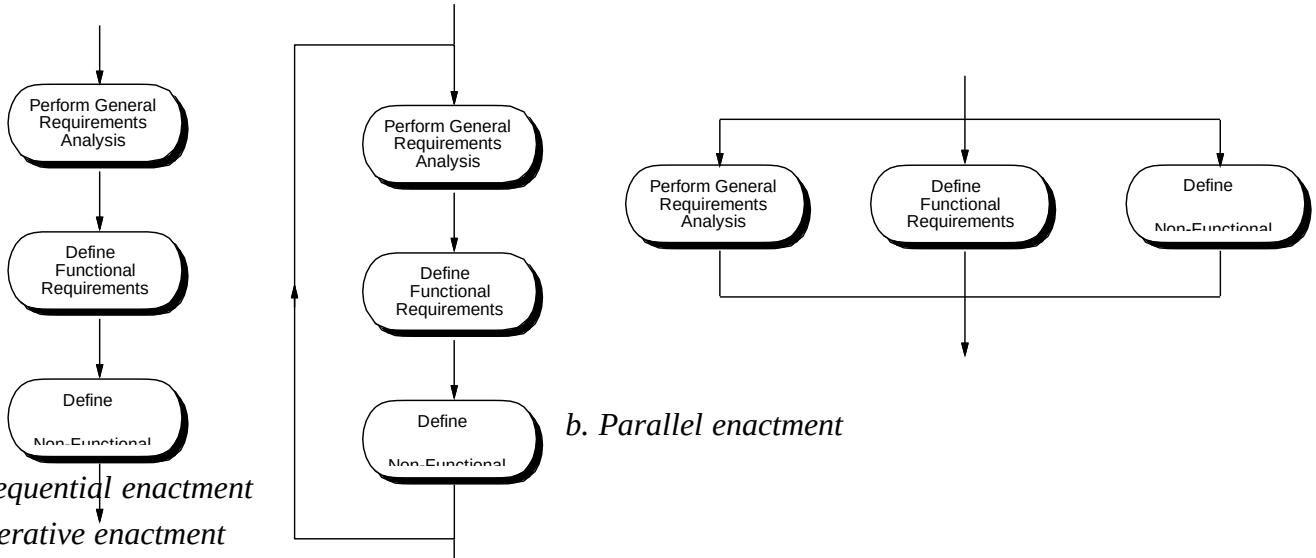
Consider the three major sub-steps of under **Analyze and Specify** in Figure 3. The normal order of these steps is as listed in the figure. This means that "Perform General Requirements Analysis", is normally performed before "Define Functional Requirements" and "Define Non-Functional Requirements". How-

ever in some cases, it may be appropriate to analyze the non-functional requirements before the other steps, or to iterate through all three of the steps in several passes. The important point is that in abstracting out a particular enactment order, the textual process depiction allows the basic structure of the process to be separated from the order of enactment.

Styles of Process Enactment

Once the steps of a software process are defined, they can be enacted in different ways. The three general forms of ordered enactment are *sequential, iterative,* and *parallel*. These are illustrated in Figure 4 for the three sub-steps of the **Analyze and Specify** step.

Sequential enactment means that the steps are performed one after the other in a strictly sequential order. A preceding step must be completed before the following step begins. For the three steps in Figure a, this means that the general analysis is completed first, followed by functional requirements, followed by non-functional requirements.



a. *Sequential enactment*

b. *Iterative enactment*

b. Parallel enactment

Iterative enactment follows an underlying sequential order, but allows a step to be only partially completed before the following step begins. Then at the end of a sequence, the steps can be re-enacted to complete some additional work. When each step is fully completed, the entire sequence is done. In Figure b, some initial work on general analysis can be completed, enough to start the function requirements analysis. After some functional requirements are done, work on the non-functional requirements can begin. Then the three steps are repeated until each is complete.

Parallel enactment allows two or more steps to be performed at the same time, independent of one another. When the work of each step is completed, the process moves on to the subsequent steps.

Which of these enactment styles to use is determined by the mutual dependencies among the steps? For some projects, the determination may be made that a complete understanding of the general requirements is necessary before the functional and non-functional requirements begin. In this case, a strictly sequential order is followed. In other projects, it may be determined that general requirements need only be partially understood initially, in which case an in iterative order is appropriate.

In this particular example that deals with analysis, a purely parallel order is probably not appropriate, since at least some understanding of the general requirements is necessary before functional and non-functional requirements are analyzed. Given this, a hybrid process order can be employed, such as shown in Figure 5. In this hybrid style of enactment, a first pass at general analysis is performed. Then the functional and nonfunctional analysis proceed in parallel. The process then iterates back to refine the general requirements and then proceed with further functional and non-functional refinements.

The three styles of process enactment discussed so far apply to process steps that are performed in some order relative to one another. A fourth kind of enactment is *pervasive*. A pervasive process step is performed continuously throughout the entire process, or at regularly scheduled points in time. A good

example of pervasive process steps are those related to project management. A well managed project will have regularly-scheduled meetings that occur on specific scheduled dates, independent of what specific ordered step developers may be conducting. The steps of the process dealing with project supervision occur essentially continuously, as the supervisors oversee developer's work, track progress, and ensure

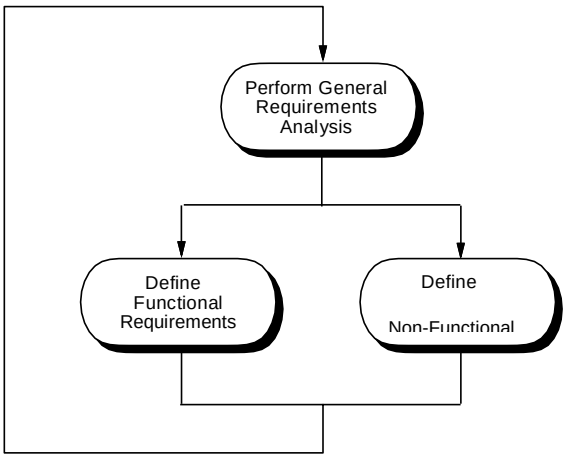


Figure 5: Hybrid process enactment.

that the process is on schedule.

Testing is another part of the software process that can be considered to be pervasive. In some traditional models of software process, testing is an ordered step that comes near the end, after the implementation is complete. The process used in this book considers testing to be a pervasive step that is conducted at regularly schedule intervals, throughout all phases of development.

The people who make the determination of which style of enactment to use are those who define the process in the first place. Process definers are generally senior technical and management staff of the development team. These are the people who understand the type of software to be developed and the capabilities of the staff who will develop it. The remaining sections of this chapter contain further discussion on the rationale for choosing different styles of process enactment, as well as different overallprocess structures.

Defining a Software Process

This book presents and follows a specific software process. The purpose of presenting a particular process is three-fold:

a. to define a process that is useful for a broad class of end-user software, including the example soft- ware system presented in the book
b. to provide an organizational framework for presenting technical subject matter
c. to give a concrete example of process definition, that can be used for guidance in defining other software processes

An important point to make at the outset is that this is "a" software process, not "the" process. There is in fact no single process that is universally applicable to all software. The process employed in this book is useful for a reasonably wide range of end-user products. However, this process, as any other, must be adapted to suit the needs of a particular development team working on a particular project. A good way to regard the process is as a representative example of process definition. Further discussion of process adaptation appears later in the chapter.

One of the most important things to say about software process is "use one that works". This means that technical details of a process and its methodologies are often less important than how well the process suits the project at hand. Above all, the process should be one that everyone thoroughly understands and can be productive using. There is no sense having management dictate a process from on high that the customers and technical staff cannot live with. The management and technical leaders who define a soft- ware process

must understand well the people who will use it, and consult with them as necessary before, during, and after the establishment of a process. In order for all this to happen, the process must be clearly defined, which is what this chapter is about.

The top-level steps of the book's process are shown in Figure 6. These steps are a refinement of the general software process presented at the beginning of the chapter in Figure 1. The refined process has the following enhancements compared to the more general one:

- the "Analyze and Specify" step has been broken down into two separate steps;
- similarly, the "Design and Implement" step has been broken into two separate steps;

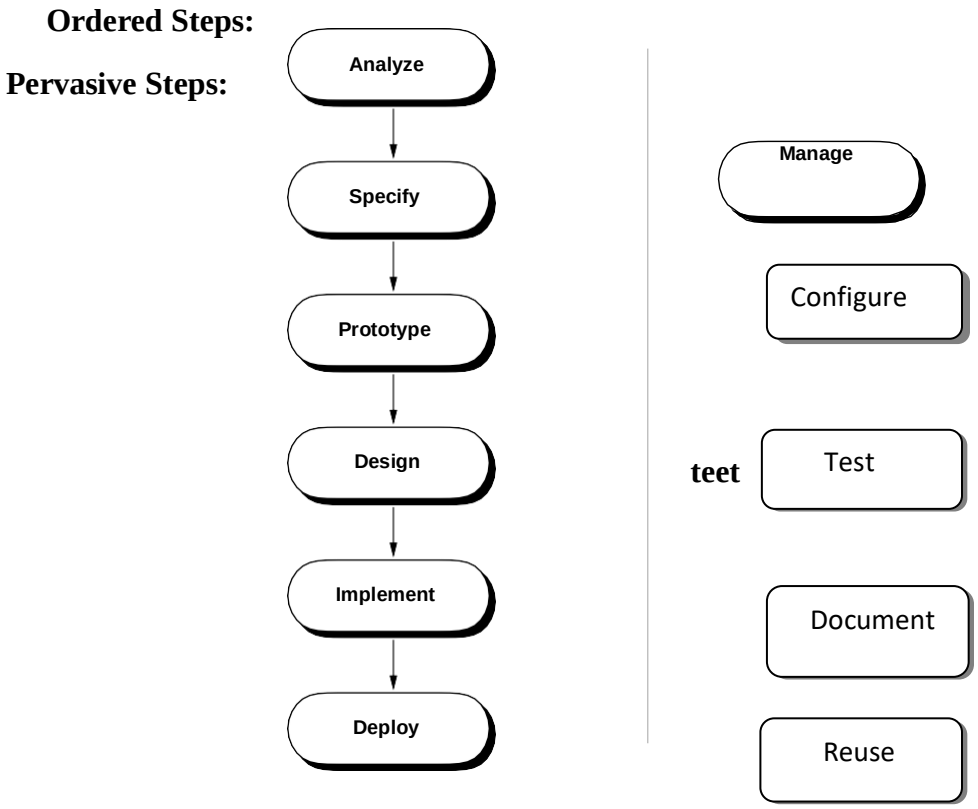


Figure 6: Top-level steps of the process used in the book.

- step names have been shortened to single words for convenient reference;
- prototyping and deployment steps have been added, details of which are discussed shortly;
- testing has been made a pervasive step of instead an ordered step following implementation; this sig- nifies that testing will be carried out at regularly scheduled points throughout the process, not just after the implementation is completed;
- Additional pervasive steps have been added for the process activities that manage the software project, configure software artifacts, document the artifacts, and reuse existing artifacts.

From a problem solving perspective, the **Analyze** and **Specify** steps taken together constitute the problem definition phase; the **Design** and **Implement** steps together comprise the problem solution phase. The new **Prototype** step is a "pre-solution", where the developers rapidly produce a version of the product with reduced functionality. The purpose of the prototype is to investigate key product features before all of the details are finished. The **Deploy** step elevates the process from one of plain problem solving to one that delivers a working product to the end users, once the implementation is completed.

The Software Process Model

A software process model is a specified definition of a software process, which is presented from a particular perspective. Models, by their nature, are a simplification, so a software process model is an abstraction of the actual process, which is being described. Process models may contain activities, which

are part of the software process, software product, and the roles of people involved in software engineering. Some examples of the types of software process models that may be produced are:

1. **A workflow model:** This shows the series of activities in the process along with their inputs, outputs and dependencies. The activities in this model perform human actions.
2. **2. A dataflow or activity model:** This represents the process as a set of activities, each of which carries out some data transformations. It shows how the input to the process, such as a specification is converted to an output such as a design. The activities here may be at a lower level than activities in a workflow model. They may perform transformations carried out by people or by computers.
3. **3. A role/action model:** This means the roles of the people involved in the software process and the activities for which they are responsible.

There are several various general models or paradigms of software developmenta Program for Beginners

1. **The waterfall approach:** This takes the above activities and produces them as separate process phases such as requirements specification, software design, implementation, testing, and so on. After each stage is defined, it is "signed off" and development goes onto the following stage.
2. **Evolutionary development:** This method interleaves the activities of specification, development, and validation. An initial system is rapidly developed from a very abstract specification.
3. **Formal transformation:** This method is based on producing a formal mathematical system specification and transforming this specification, using mathematical methods to a program. These transformations are 'correctness preserving.' This means that you can be sure that the developed programs meet its specification.
4. **System assembly from reusable components:** This method assumes the parts of the system already exist. The system development process target on integrating these parts rather than developing them from scratch

➢ Software Process Models

Prescriptive process models define a set of activities, actions, tasks, milestones, and work products that are required to engineer high-quality software. These process models are not perfect, but they do provide a useful roadmap for software engineering work.

A prescriptive process model populates a process framework with explicit task sets for software engineering actions.

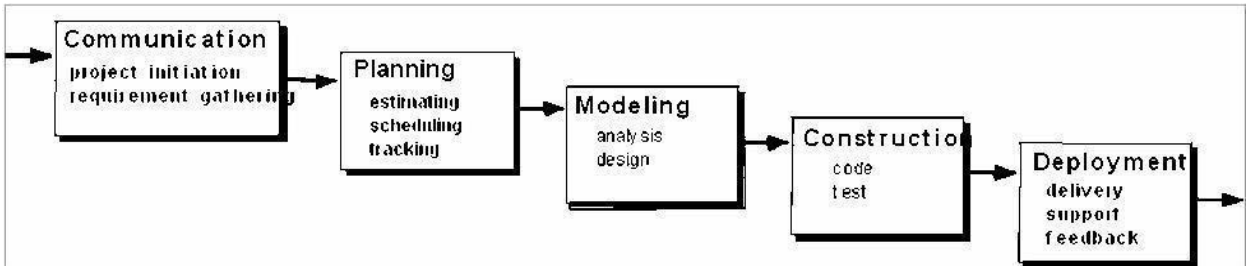
THE WATERFALL MODEL:

The waterfall model, sometimes called the *classic life cycle*, suggests a systematic sequential approach to software development that begins with customer specification of requirements and progresses through planning, modeling, construction, and deployment.

Context: Used when requirements are reasonably well understood.

Advantage:

It can serve as a useful process model in situations where requirements are fixed and work is to proceed to complete in a linear manner.



The **problems** that are sometimes encountered when the waterfall model is applied are:

Real projects rarely follow the sequential flow that the model proposes. Although the linear model can accommodate iteration, it does so indirectly. As a result, changes can cause confusion as the project team proceeds.

It is often difficult for the customer to state all requirements explicitly. The waterfall model requires this and has difficulty accommodating the natural uncertainty that exist at the beginning of many projects.

The customer must have patience. A working version of the programs will not be available until late in the project time-span. If a major blunder is undetected then it can be disastrous until the program is reviewed.

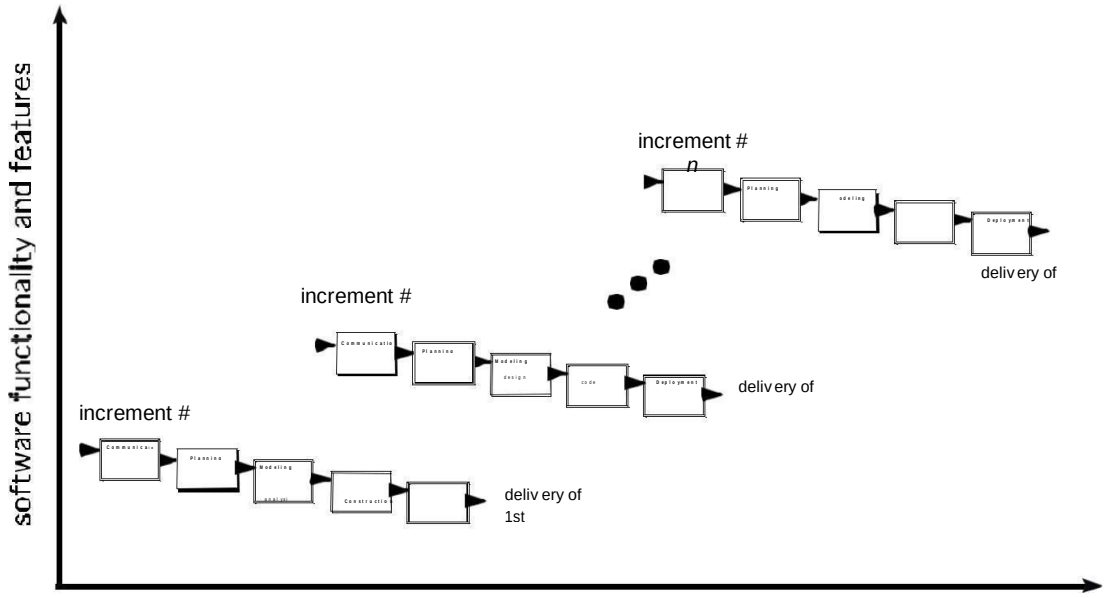
INCREMENTAL PROCESS MODELS:

The incremental model

The RAD model

THE INCREMENTAL MODEL:

Context: Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project. Early increments can be implemented with fewer people. If the core product is well received, additional staff can be added to implement the next increment. In addition, increments can be planned to manage technical risks.



The incremental model combines elements of the waterfall model applied in an iterative fashion.

The incremental model delivers a series of releases called increments that provide progressively more functionality for the customer as each increment is delivered.

When an incremental model is used, the first increment is often a core product. That is, basic requirements are addressed. The core product is used by the customer. As a result, a plan is developed for the next increment.

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.

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.

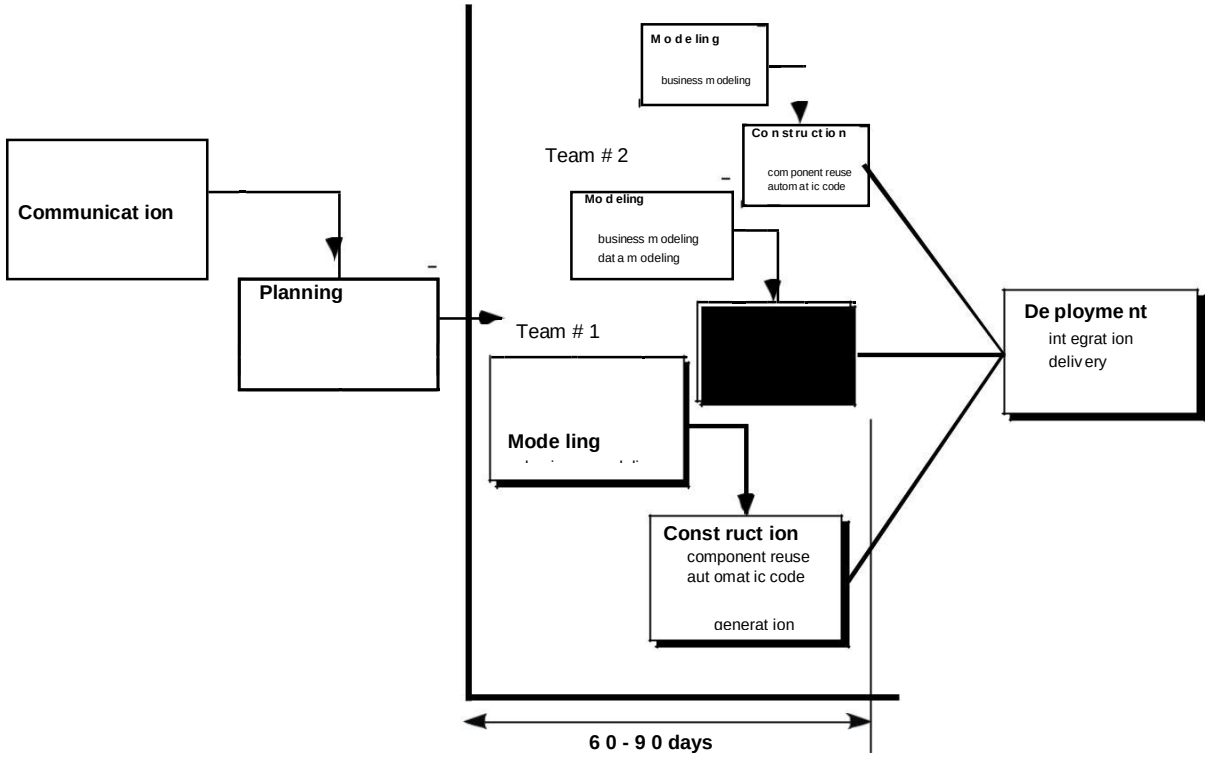
Difference: The incremental process model, like prototyping and other evolutionary approaches, is iterative in nature. But unlike prototyping, the incremental model focuses on delivery of an operational

product with each increment.

THE RAD MODEL:

Rapid Application Development (RAD) is an incremental software process model that emphasizes a short development cycle. The RAD model is a –high-speed‖ adaption of the waterfall model, in which rapid development is achieved by using a component base construction approach.

Context: If requirements are well understood and project scope is constrained, the RAD process enables a development team to create a –fully functional system‖ within a very short time period.



The RAD approach maps into the generic framework activities.

Communication works to understand the business problem and the information characteristics that the software must accommodate.

Planning is essential because multiple software teams works in parallel on different system functions.

Modeling encompasses three major phases- business modeling, data modeling and process modeling- and establishes design representation that serve existing software components and the application of automatic code generation.

Deployment establishes a basis for subsequent iterations. The RAD approach has **drawbacks**:

For large, but scalable projects, RAD requires sufficient human resources to create the right number of RAD teams.

If developers and customers are not committed to the rapid-fire activities necessary to complete the system in a much abbreviated time frame, RAD projects will fail

If a system cannot be properly modularized, building the components necessary for RAD will be problematic

If high performance is an issue, and performance is to be achieved through tuning the interfaces to system components, the RAD approach may not work; and

RAD may not be appropriate when technical risks are high.

➢ Evolutionary process models:

Evolutionary process models produce with each iteration produce an increasingly more complete version of the software with every iteration.

Evolutionary models are iterative. They are characterized in a manner that enables software engineers to develop increasingly more complete versions of the software.

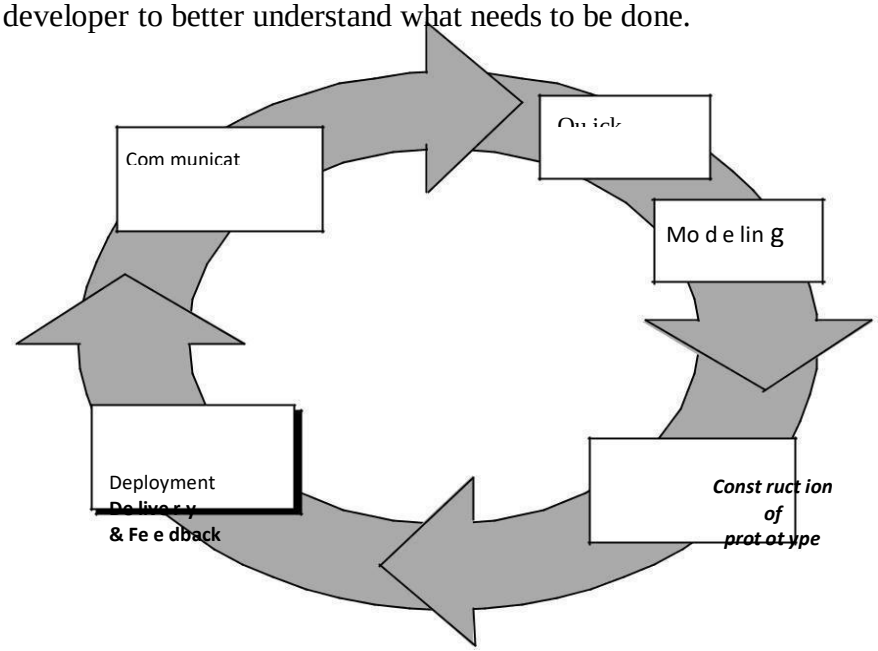
PROTOTYPING:

Prototyping is more commonly used as a technique that can be implemented within the context of anyone of the process model.

The prototyping paradigm begins with communication. The software engineer and customer meet and define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory.

Prototyping iteration is planned quickly and modeling occurs. The quick design leads to the construction of a prototype. The prototype is deployed and then evaluated by the customer/user.

Iteration occurs as the prototype is tuned to satisfy the needs of the customer, while at the same time enabling the developer to better understand what needs to be done.



Context:

If a customer defines a set of general objectives for software, but does not identify detailed input, processing, or output requirements, in such situation *prototyping* paradigm is best approach.

If a developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system then he can go for this *prototyping* method.

Advantages:

The prototyping paradigm assists the software engineer and the customer to better understand what is to be built when requirements are fuzzy.

The prototype serves as a mechanism for identifying software requirements. If a working prototype is built, the developer attempts to make use of existing program fragments or applies tools.

Prototyping can be **problematic** for the following reasons:

The customer sees what appears to be a working version of the software, unaware that the prototype is held together ―with chewing gum and baling wire‖, unaware that in the rush to get it working we haven‘t considered overall software quality or long-term maintainability. When informed that the product must be rebuilt so that high-levels of quality can be maintained, the customer cries foul and demands that ‒a few fixes‖ be applied to make the prototype a working product. Too often, software development relents.

The developer often makes implementation compromises in order to get a prototype working quickly. An inappropriate operating system or programming language may be used simply because it is available and known; an inefficient algorithm may be implemented simply to demonstrate capability. After a time, the

developer may become comfortable with these choices and forget all the reasons why they were inappropriate. The less-than-ideal choice has now become an integral part of the system.

THE SPIRAL MODEL

The spiral model, originally proposed by Boehm, 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.

Using the spiral model, software is developed in a series of evolutionary releases. During early iterations, the release might be a paper model or prototype. During later iterations, increasingly more complete versions of the engineered system are

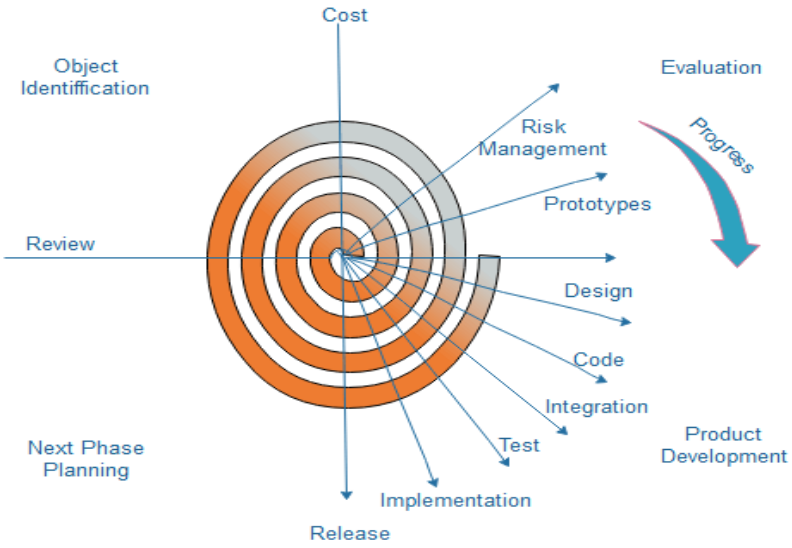


Fig. Spiral Model

Context: The spiral model can be adopted to apply throughout the entire life cycle of an application, from concept development to maintenance.

Advantages:

It provides the potential for rapid development of increasingly more complete versions of the software.

The spiral model is a realistic approach to the development of large-scale systems and software. The spiral model uses prototyping as a risk reduction mechanism but, more importantly enables the developer to apply the prototyping approach at any stage in the evolution of the product.

Draw Backs:

The spiral model is not a panacea. It may be difficult to convince customers that the evolutionary approach is controllable. It demands considerable risk assessment expertise and relies on this expertise for success. If a major risk is not uncovered and managed, problems will undoubtedly occur.

➢ Component-Based Development (CBD)

Component-based development (CBD) is a procedure that accentuates the design and development of computer-based systems with the help of reusable software components. With CBD, the focus shifts from software programming to software system composing

Component-based development techniques involve procedures for developing software systems by choosing ideal off-the-shelf components and then assembling them using a well-defined software architecture. With the systematic reuse of coarse-grained components, CBD intends to deliver better quality and output

Component-based development is also known as component-based software engineering (CBSE).

The key goals of CBD are as follows

Save time and money when building large and complex systems: Developing complex software systems with the help of off-the-shelf components helps reduce software development time substantially. Function points or similar techniques can be used to verify the affordability of the existing method.

- Enhance the software quality: The component quality is the key factor behind the enhancement of software quality.
- Detect defects within the systems: The CBD strategy supports fault detection by testing the components; however, finding the source of defects is challenging in CBD.

Some advantages of CBD include

Minimized delivery:

- Search in component catalogs
- Recycling of pre-fabricated components
- Improved efficiency:
- Developers concentrate on application development
- Improved quality:
- Component developers can permit additional time to ensure quality
- Minimized expenditures

The specific routines of CBD are

Component development

- Component publishing
- Component lookup as well as retrieval
- Component analysis
- Component assembly

➢ Formal Methods Model

Definition

The formal methods model is an approach to software engineering that applies mathematical methods or techniques to the process of developing complex software systems. The approach uses a formal specification language to define each characteristic of the system. The language is very particular and employs a unique syntax whose components include objects, relations, and rules.

When used together, these components can validate the correctness of each characteristic. Think of the process like balancing a series of equations. At each step in the series, if the right-side of the equation doesn't equal the left, there's a problem that must be addressed.

Formal Methods Model: Steps

There are two steps that comprise the formal methods model. Those steps are the property-based specification and the model-based specification.

Property-Based Specification

Property-based specification describes two main elements in the system; those elements are the operations that can be performed on the system and the relationships between the operations. For example, consider a simple instant messaging application for your cell phone. Then some operations might be:

- Start up
- Send message
- Receive message
- Display message, and
- Shut down

The relationships between these operations might include:

- Startup must come before any other operation.
- Shut down must be the last operation performed.
- Display message comes during each send message and after each receive message.

Model-Based Specification

Model-based specification describes the states the system can be in and how the operations can transition the system from state to state. Consider the instant messaging application example mentioned earlier. States the system may be in might include the very similar-sounding states:

- Starting up
- Sending message
- Receiving message
- Displaying message, and
- Shutting down

As for transitions, they might include:

- Clicking the application icon to enter starting up
- Or, pressing the send button to leave the sending message state

Please note that these steps were described in familiar language rather than through the specification syntax for clarity's sake. With a formal specification, a non-mathematical, natural language description is transcribed into a formal language, which allows it to be rigorously checked for validity. As an example, the actual syntax to describe an instant message connection might look something like:

- message_connection: M System Connections
- assigned to: Instant Connections: System Connections
- available, sending, receiving: M System Connections

➢ Software Engineering- Fourth Generation Techniques

Implementation using a **4GL** (4th Generation Techniques) enables the software developer to represent desired results in a manner that leads to automatic generation of code to create those results. Obviously, a data structure with relevant information must exist and be readily accessible by the 4GL. To transform a 4GT implementation into a product, the developer must conduct thorough testing, develop meaningful documentation, and perform all other solution integration activities that are required in other software engineering paradigms. In addition, the 4GT developed software must be built in a manner that enables maintenance to be performed expeditiously.

Software development environment that supports the 4GT paradigm includes some or all of the following tools

1) Non-procedural languages for database query
2) Report generation
3) Data manipulation
4) Screen interaction and definition
5) Code generation and High-level graphics capability
6) Spreadsheet capability
7) Automated generation of HTML and similar languages used for Web-site creation using advanced software tools

Pros and Cons Proponents claim dramatic reduction in software development time and greatly improved productivity for people who build software. Opponents claim that current 4GT tools are not all that much easier to use than programming languages, that the resultant source code produced by such tools is "inefficient" and that the maintainability of large software systems developed using 4GT is open to question.

Advantages:

Simplified the programming process.

Use non-procedural languages that encourage users and programmers to specify the results they want, while the computers determines the sequence of instruction that will accomplish those results.

Use natural languages that impose no rigid grammatical rules.

Disadvantages:

Less flexible that other languages

Programs written in 4GLs are generally far less efficient during program execution that programs in high-level languages.
Therefore, their use is limited to projects that do not call for such efficiency.

➢ An Agile view of processes and Development:

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.

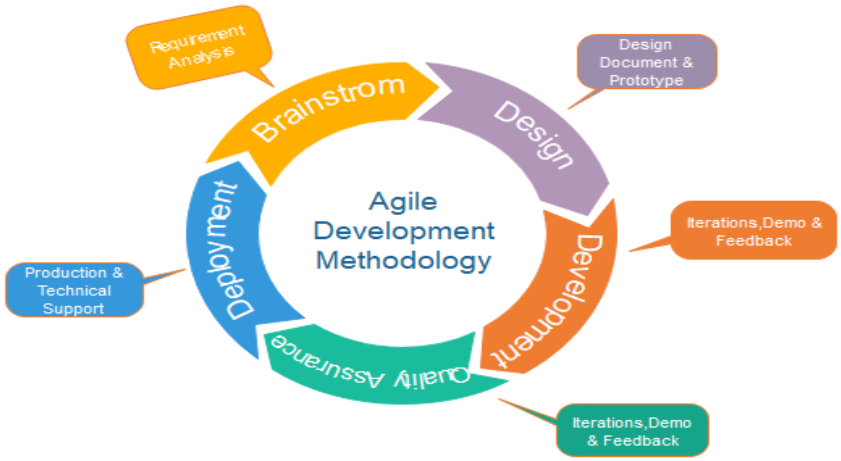


Fig. Agile Model

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)

Scrum

SCRUM is an agile development process focused primarily on ways to manage tasks in team-based development conditions.

There are three roles in it, and their responsibilities are:

- **Scrum Master:** The scrum can set up the master team, arrange the meeting and remove obstacles for the process
- **Product owner:** The product owner makes the product backlog, prioritizes the delay and is responsible for the distribution of functionality on each repetition.
- **Scrum Team:** The team manages its work and organizes the work to complete the sprint or cycle.

Extreme Programming(XP)

This type of methodology is used when customers are constantly changing demands or requirements, or when they are not sure about the system's performance.

Crystal:

There are three concepts of this method-

1. Chartering: Multi activities are involved in this phase such as making a development team, performing feasibility analysis, developing plans, etc.
2. Cyclic delivery: under this, two more cycles consist, these are:

A. Team updates the release plan.

B. Integrated product delivers to the users.

3. Wrap up: According to the user environment, this phase performs deployment, post-deployment.

Dynamic Software Development Method(DSDM):

DSDM is a rapid application development strategy for software development and gives an agile project distribution structure. The essential features of DSDM are that users must be actively connected, and teams have been given the right to make decisions. The techniques used in DSDM are:

1. Time Boxing
2. MoSCoW Rules
3. Prototyping

The DSDM project contains seven stages:

1. Pre-project
2. Feasibility Study
3. Business Study
4. Functional Model Iteration
5. Design and build Iteration
6. Implementation
7. Post-project

Feature Driven Development (FDD):

This method focuses on "Designing and Building" features. In contrast to other smart methods, FDD describes the small steps of the work that should be obtained separately per function.

Lean Software Development:

Lean software development methodology follows the principle "just in time production." The lean method indicates the increasing speed of software development and reducing costs. Lean development can be summarized in seven phases.

1. Eliminating Waste
2. Amplifying learning
3. Defer commitment (deciding as late as possible)
4. Early delivery
5. Empowering the team
6. Building Integrity
7. Optimize the whole

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.

Advantage (Pros) of Agile Method:

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

Disadvantages (Cons) of Agile Model:

1. Due to the shortage of formal documents, it creates confusion and crucial decisions taken throughout various phases can be misinterpreted at any time by different team members.
2. Due to the lack of proper documentation, once the project completes and the developers allotted to another project, maintenance of the finished project can become a difficulty.

➢ Software Engineering Practices in Software Development

A software engineer applies engineering practices to software development, and typically handles the overall system design of the software application. While some software engineers also handle programming, these engineers typically hand off the engineered designs to software programmers for coding. Once the coding is complete, they test the software and make sure it performs within the engineered requirements.

Software engineering is divided into several specializations, which focus on a particular software development area. A few of the most common software engineering specialties include:

- Requirements engineering looks at the overall requirements of software, how it fits into the current system, and whether the code from software programmers meets the necessary requirements. Software quality management fulfills a similar set of duties, although these software engineers may only be confirming the software meets someone else's set of requirements.

Software engineering process looks at the software development lifecycle and identifies areas of improvement and optimization. By engineering the development lifecycle itself, this software engineer ensures the entire development process runs more efficiently.

Software design covers one of the most common software engineering specialities. These engineers define

everything from the software's infrastructure to its interface before programmers start on the code, in order to create software effective for the environment it's getting deployed in.
The typical software engineer has many duties and plays an integral part in the software design and development process:

- Look at proposed software development projects and determines whether the guidelines follow sound engineering practices. If they don't, the engineer addresses any problems and provides an updated project requirements list.
  Creates documentation for the software design and development of the project, which allows the programmers to work effectively and any other engineers working on the project to understand what's going on with the design.
- Examines business needs to determine whether software can solve existing problems, fix operational inefficiencies, or otherwise optimize work processes. When these areas are identified, the software engineer begins working on a software design document to see whether the solution is feasible with the resources on hand.
- Software quality assurance and testing sometimes fall in the software engineering realm, especially if the engineer created the original design document.
  Software engineers create needed efficiency in software development projects, which ensures the projects fit the needs of the company, stay within the project scope, and tests developed software to confirm it follows the engineering practices laid out in the initial design.

Software Engineering:

Software engineering is defined as a process of analyzing user requirements and then designing, building, and testing software application which will satisfy those requirements.
Let's look at the various definitions of software engineering:

- IEEE, in its standard 610.12-1990, defines software engineering as the application of a systematic, disciplined, which is a computable approach for the development, operation, and maintenance of software.
- Fritz Bauer defined it as 'the establishment and used standard engineering principles. It helps you to obtain, economically, software which is reliable and works efficiently on the real machines'.
- Boehm defines software engineering, which involves, 'the practical application of scientific knowledge to the creative design and building of computer programs. It also includes associated documentation needed for developing, operating, and maintaining them.

➢ Software Engineering Practice

Practice is a broad array of concepts, principles, methods, and tools that you must consider as software is planned and developed.
It represents the details—the technical considerations and how to's—that are below the surface of the software process—the things that you'll need to actually build high-quality computer software.

The Essence of Practice

This section lists the generic framework (communication, planning, modeling, construction, and deployment) and umbrella (tracking, risk management, reviews, measurement, configuration management, reusability management, work product creation, and product) activities found in all software process models.

1. **Understand the problem** (communication and analysis).
   Who are the stakeholders?
   What are the unknowns? "Data, functions, features to solve the problem?"
   Can the problem be compartmentalized? "Smaller that may be easier to understand?
   Can the problem be represented graphically? Can an analysis model be created?
2. **Plan a solution** (modeling and software design).
   Have you seen a similar problem before?
   Has a similar problem been solved? If so, is the solution reusable?
   Can sub-problems be defined?

Can you represent a solution in a manner that leads to effective implementation?

3.Carry out the plan (code generation).

Does the solution conform to the plan?

Is each component part of the solution probably correct?

4.Examine the result for accuracy (testing and quality assurance).

Is it possible to test each component part of the solution?

Does the solution produce results that conform to the data, functions, features, and behavior that are required?

1.Core Principles

The Reason It All Exists: Provide value to the customer and the user. If you can't provide value, then don't do it.

KISS—Keep It Simple, Stupid! *All design should be as simple as possible, but no simpler.* This facilitates having a more easily understood and easily maintained system.

Maintain the product and project "vision." *A clear vision is essential to the success of a S/W project.*

What you produce, others will consume. *Always specify, design, and implement knowing someone else have to understand what you are doing.*

Be open to the Future. *Never design yourself into a corner*. Always ask "what if," and prepare yourself for all possible answers by creating systems that solve the general problem, not just the specific one.

Plan Ahead for Reuse. *Planning ahead for reuse reduces the cost and increases the value of both the reusable components and the systems into which they are incorporated.*

Think! Placing clear, complete thought before action almost always produces better results.

➢ Communication Practices

Before customer requirements can be analyzed, modeled, or specified they must be gathered through a *communication* (also called *requirement elicitation*) activity.

Effective communication (among technical peers, with the customer and other stakeholders, and with project managers) is among the most challenging activities that confront a S/W engineer.

In this context, the following are communication **principles** and concepts that apply to customer communication:

Listen: focus on the speaker's words, rather than formulating your response to those words. Be a polite listener.

Prepare before you communicate: Spend the time to understand the problem before you meet with others "research".

Someone should facilitate the communication activity. Have a leader "moderator" to keep the conversation moving in a productive direction.

Face-to-face communication is best.

Take notes and document decisions.

Collaborate with the customer. Each small collaboration serves to build trust among team members and creates a common goal for the team.

Stay focused, modularize your discussion. The facilitator should keep the conversation modular; leaving one topic only after it has been resolved.

Draw pictures when things are unclear.

(a)Once you agree to something, move on; (b) if you can't agree to something, move one; (c) if a feature or function is unclear and can't be clarified at the moment, move on.

➢ Planning Practices

The planning activity encompasses a set of management and technical practices that enable the S/W team to define a road map as it travels toward its strategic goal and tactical objectives.

Regardless of the rigor with which planning is conducted, the following principles always apply:

Understand the project scope. Scope provides the S/W team with a destination.

Involve the customer (and other stakeholders) in the planning activity. The customer defines priorities and establishes project constraints. S/W engineers must often negotiate order of delivery, timelines, and other related issues.
Recognize that planning is iterative. A plan must be adjusted to accommodate changes.
Estimate based on what you know. The intent of estimation is to provide an indication of effort, cost, and task duration, based on the team's current understanding of the work to be done.
Consider risk as you define the plan.
Be realistic. Even the best S/W engineers make mistakes.
Adjust granularity as you plan. A fine granularity plan provides significant work task detail that is planned over relatively short time increments. A coarse granularity plan provides broader work tasks that are planned over longer time periods.
Define how quality will be achieved.
Define how you'll accommodate changes. "Can the customer request a change at any time?"
Track what you've planned and make adjustments as required.
Barry Boehm states: "You need an organizing principle that scales down to provide simple plans for simple projects."
Boehm suggests an approach that addresses project objectives, milestones, and schedules, responsibilities, management and technical approaches, and required resources.
Boehm calls it *W^5HH principle*, after a series of questions that lead to a definition of key project characteristics and the resultant project plan.
Why is the system being developed? Does the business purpose justify the expenditure of people, time and money?
What will be done? Identify the functionality to be built.
When will it be accomplished? Establish a workflow and timeline for key project tasks and identify milestones required by the customer.
Who is responsible for a function? Define members' roles and responsibilities.
Where are they located (organizationally)? Customers also have responsibilities.
How will the job be done technically and managerially? Once a scope is defined, a technical strategy must be defined.
How much of each resource is needed? The answer is derived by developing estimates based on answers to earlier questions.

➢ Modeling Practices

The process of developing analysis and design models is described in this section. The emphasis is on describing how to gather the information needed to build reasonable models, but no specific modeling notations are presented in this chapter. UML and other modeling notations are described in detail later in the text.
In S/W Eng. work, two models are created: analysis models and design models.
Analysis models represent the customer requirements by depicting the S/W in three different domains: the information domain, the functional domain, and the behavioral domain.
Design models represent characteristics of the S/W that help practitioners to construct it effectively: the architecture, the user interface, and component-level detail.

Analysis Modeling Principles

The information domain of a problem must be represented and understood. The *information domain* encompasses the data that flow into the system (end-users, other systems, or external devices), the data that flow out of the system and the data stores that collect and organize persistent data objects.
Represent software functions. Functions can be described at many different levels of abstraction, ranging from a general statement of purpose to a detailed description of the processing elements that must be invoked.

Represent software behavior. The behavior of the S/W is driven by the interaction with the external environment.
The models that depict information, function, and behavior must be partitioned in a manner that uncovers detail in a layered fashion (or hierarchical).
The analysis task should move from essential information toward implementation detail. Analysis begins by describing the problem from the end-user perspective. The"essence" of the problem is described without any consideration of how a solution will be implemented.
Design Modeling Principles
The software design model is the equivalent of an architect's plans for a house.
Set of principles used:
Design must be traceable to the analysis model. The analysis model describes the information domain of the problem, user visible functions, system behavior, and a set of analysis classes that package business objects with the methods that service them.
The design model translates this information into an architecture: a set of subsystems that implement major functions, and a set of component-level designs that are the realization of analysis class.
Always consider architecture. S/W architecture is the skeleton of the system to be built. Only after the architecture is built should the component-level issues should be considered.

Focus on the design of data as it is as important as a design. Data design is an essential element of t architectural design.
Interfaces (both user and internal) must be designed. A well designed interface makes integration easier and assists the tester in validating component functions.
User interface design should be tuned to the needs of the end-user. "Ease of use."
Component-level design should exhibit functional independence. The functionality that is delivered by a component should be *cohesive*- that is, it should focus on one and only one function.
Components should be loosely coupled to one another and to the external environment. *Coupling* is achieved in many ways – via a component interface, by messaging through global data. Coupling should be kept as low as is reasonable. As the level of coupling increases, error propagation also increases and the overall maintainability of the system decreases.
Design representation (models) should be easily understood.
The design model should be developed iteratively. With each iteration, the designer should strive for greater simplicity.

➢ Construction Practices

In this text "construction" is defined as being composed of both coding and testing. The purpose of testing is to uncover defects. Exhaustive testing is not possible so processing a few test cases successfully does not guarantee that you have bug free program. Unit testing of components and integration testing will be discussed in greater later in the text along with software quality assurance activities.

Although testing has received increased attention over the past decade, it is the weakest part of software engineering practice for most organizations.

Coding Principles and Concepts
Preparation Principles: Before writing one line of code, be sure of:

1. Understand the problem you are trying to solve.
2. Understand the basic design principles.
3. Pick a programming language that meets the needs of the S/W to be built and the environment in which it will operate.
4. Select a programming environment that provides tool that will make your work easier.
5. Create a set of unit tests that will be applied once the component you code is completed.

Coding Principles: As you begin writing code, be sure you
1.Constrain your algorithm by following structured programming practice.

2.Select the proper data structure.

3.Understand the software architecture.

4.Keep conditional logic as simple as possible.

5.Create easily tested nested loops.

6. Write code that is self-documenting.

7. Create a visual layout.

Validation Principles: After you've completed your first coding pass, be sure you

1. Conduct a code walkthrough.
2. Perform unit test and correct errors.
3. Refactor the code.

Testing Principles

- Testing is a process of executing a program with the intent of finding errors.
- A good test is one that has a high probability of finding an as-yet undiscovered error.
- A successful test is one that uncovers an as-yet-undiscovered error.

➢ Deployment Practices

Software development involves the process of creating applications and software programs by writing and maintaining the source code. It is about the complete process and the stages involved throughout the software development life cycle (SDLC).

The software development is a step by step process of inventing, specifying, coding, recording, testing and fixing bugs that is done to create and manage frameworks, software components or even to develop a complete application.

Customer Expectations for the software must be managed. "Don't promise more than you can deliver."

A complete delivery package should be assembled and tested.

A support regime must be established before the software is delivered.

Appropriate instructional materials must be provided to end-users.

Buggy software should be fixed first, delivered later.

Best Deployment in Software Practices

Implement a deployment checklist

Set up a process while you deploy a new software. A checklist helps you to follow what must be done next so that you will not miss out any of the crucial steps

Choose the right Deployment Method

Implement the software that is easy to integrate with the existing local applications and other tools.

Automated Software Deployment Process

Deployment of new versions of software manually is a daunting task that brings in a lot of possibilities of human errors. Automating the deployment process, mitigates the possibilities of errors, increases the deployment speed and streamlines the process

Adopt continuous delivery

Adopting Continuous Delivery ensures to enable the code for the required deployment. This is done by implementing the application in a proto-type environment to ensure if the application is good to function and meet the demands once deployed.

Use a continuous integration server

Continuous Server Integration is crucial for any successful agile deployment. This ensures that the developed program works on a developer's machine while the it helps you deny "integration hell".