

# **Course Name : Software Engineering and Project Management**

**Course Code : 310243**

**Credit : Th-03**

**Examination Scheme:**

**In-Sem(Paper) : 30 Marks**

**End-Sem(Paper) : 70 Marks**

**In-sem Exam :**

**End-Sem Exam :**

# Prerequisite & Course Objectives

**Prerequisite:** Fundamentals of Programming Languages

## **Course Objectives:**

- ✓ To learn and understand the principles of Software Engineering
- ✓ To be acquainted with methods of capturing, specifying, visualizing and analyzing software requirements.
- ✓ To apply Design and Testing principles to S/W project development.
- ✓ To understand project management through life cycle of the project.
- ✓ To understand software quality attributes.

# Course : Software Engineering and Project Management

## Unit 1

### **Introduction to Software Engineering, Software Process Models**

# Syllabus

**Software Engineering Fundamentals :** Nature of Software , Software Engineering Principles, The Software Process , Software Myths

**Process Models :** A Generic Process Model, Prescriptive Process Models: The Waterfall, Incremental Process(RAD), Evolutionary Process, Unified Process, Concurrent.

**Advanced Process Models & Tools :** Agile software development: Agile methods, Plan-driven and agile development, Extreme programming Practices, Testing in XP, Pair programming. Introduction to agile tools: JIRA, Kanban

**Case Studies:** An information system (mental health-care system), wilderness weather system

# What is it ?



Computer software is the product that software professionals build and then support over the long term.



It encompasses (keeps) **programs** that execute within a computer of any size and architecture, content that is presented as the computer programs execute, and descriptive information in both hard copy and virtual forms that encompass virtually any electronic media.



Software engineering encompasses a process, a collection of **methods** (practice) and an array of **tools** that allow professionals to build high-quality computer software.

# What does it ?



Software engineers build and support software, and virtually everyone in the industrialized world uses it either directly or indirectly.

# Why it is important ?



Software is important because it affects nearly every aspect of our lives and has become pervasive in our commerce, our culture, and our everyday activities.



Software engineering is important because it enables us to build complex systems in a timely manner and with high quality.

# What are the steps ?



You build computer software like you build any successful product, by applying an agile, adaptable process that leads to a high-quality result that meets the needs of the people who will use the product.



You apply a software engineering approach.

# What is the work product?



From the point of view of a software engineer, the work product is the set of programs, content (data), and other work products that are computer software.



But from the user's viewpoint, **the work product is the resultant information that somehow makes the user's world better.**

# Software Engineering Fundamentals : Nature of Software





Today, **software takes on a dual role**. It is a product, and at the same time, the vehicle for delivering a product.



As a **product**, it delivers the computing potential embodied by computer hardware or more broadly, by a network of computers that are accessible by local hardware.



Whether it **resides within a mobile phone or operates inside a mainframe computer**, software is an information transformer producing, managing, acquiring, modifying, displaying, or transmitting information that can be as simple as a single bit or as complex as a multimedia presentation derived from data acquired from dozens of independent sources.



As the **vehicle** used to deliver the product, software acts as the basis for the control of the computer (**operating systems**), the communication of information (**networks**), and the creation and control of other programs (**software tools** and **environments**)



Software delivers the most important product of our time information.



It **transforms personal data** (e.g., an individual's financial transactions) so that the data can be more useful in a local context; it **manages business information to enhance competitiveness**; it provides a gateway to worldwide information networks (e.g., the **Internet**), and provides the means for acquiring information in all of its forms.

# How should we define software ?



Software is:

- (1) instructions (computer programs) that when executed, it 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.



**Therefore, software has characteristics that are considerably different than those of hardware :-**

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

- Although some similarities exist between software development and hardware manufacturing, the two activities are fundamentally different.
- In both activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems that are nonexistent (or easily corrected) for software.
- Both activities are dependent on people, but the relationship between people applied and work accomplished is entirely different.
- Both activities require the construction of a “product,” but the approaches are different.
- Software costs are concentrated in engineering.
- This means that software projects cannot be managed as if they were manufacturing projects.

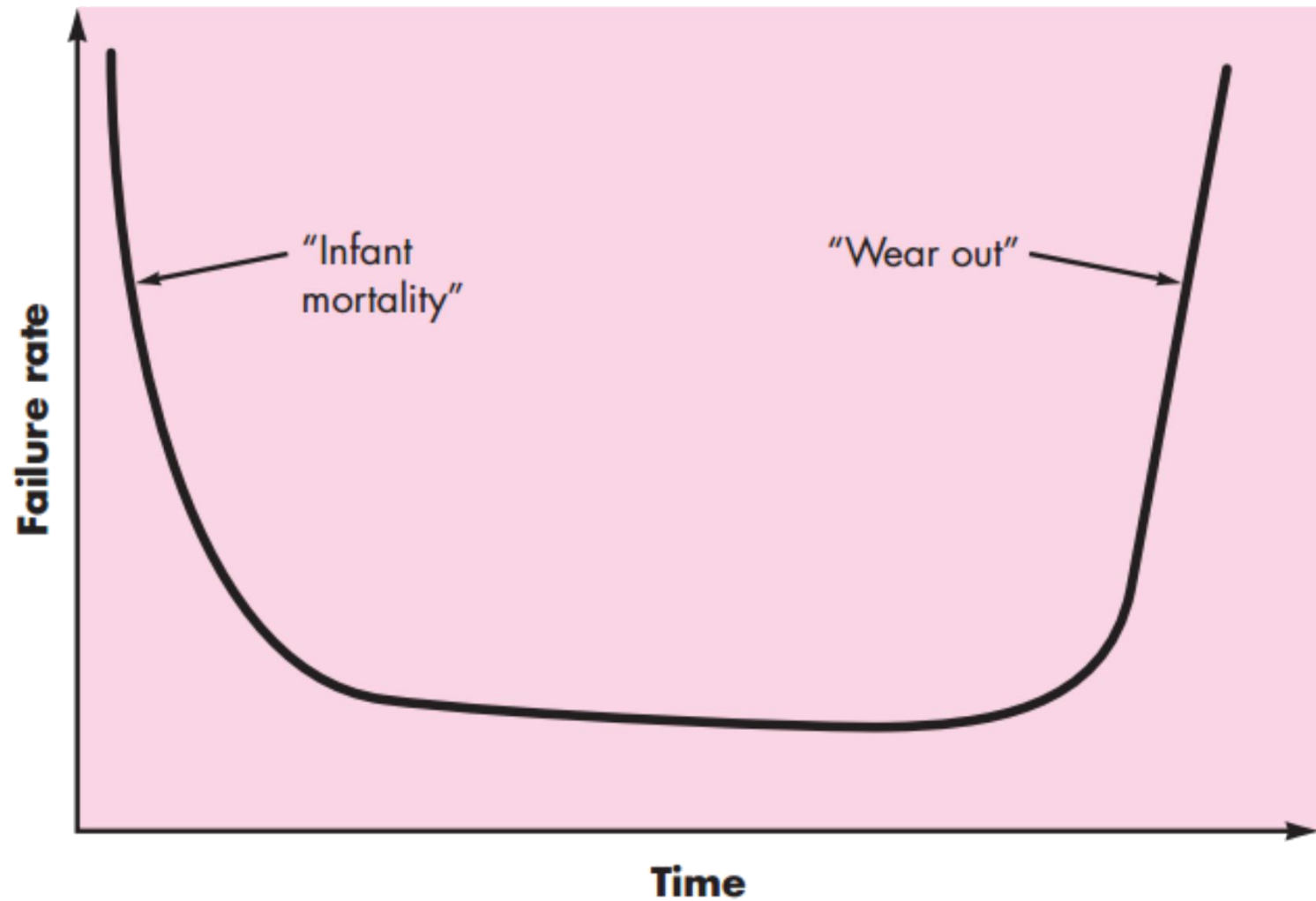


Figure 1.1 :- Failure curve for Hardware

➤ 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.

➤ 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.

## 2. Software doesn't "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.

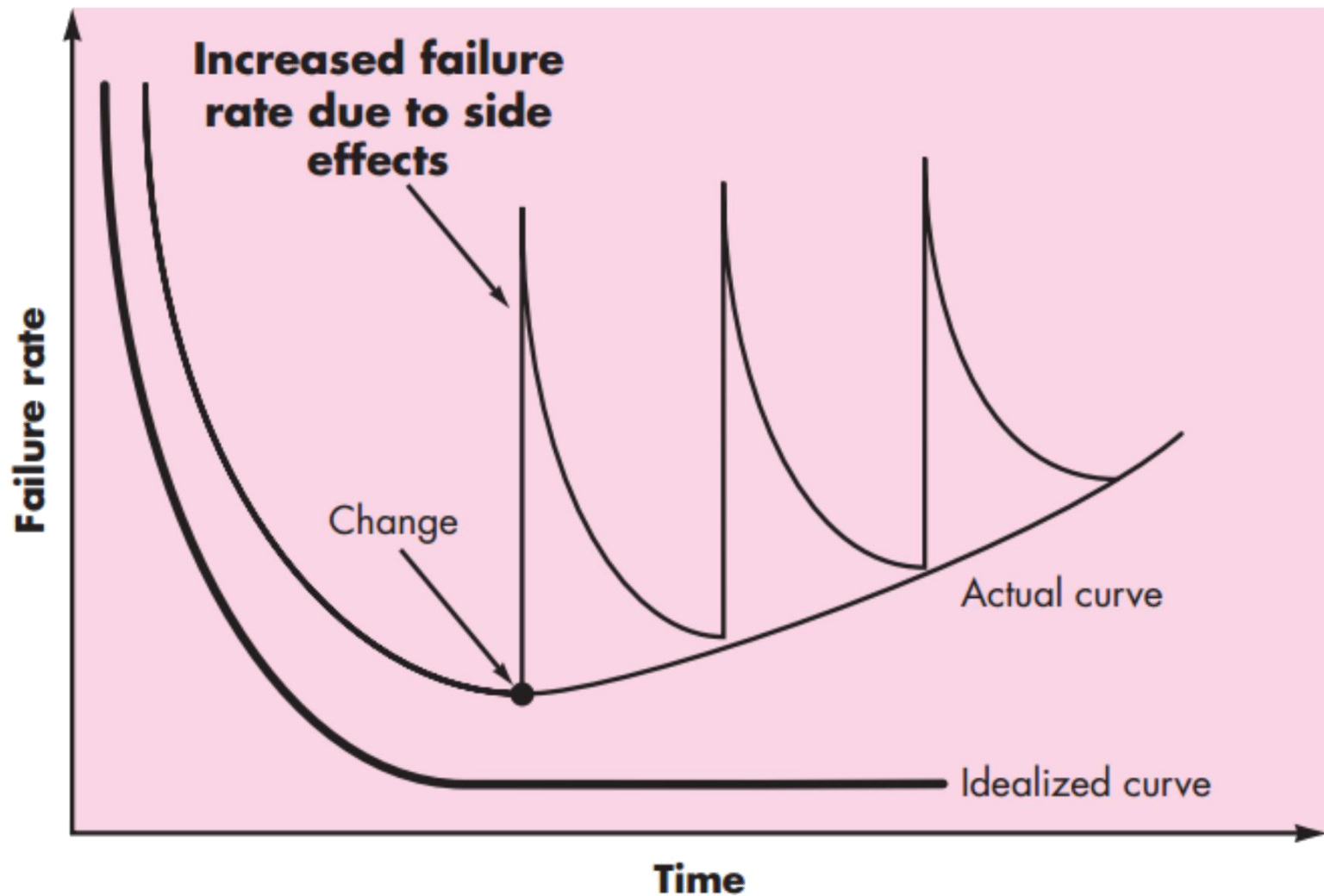


Figure 1.2 :- Failure curves for Software



➤ The idealized curve is a gross oversimplification of actual failure models for software.

➤ However, the implication is clear—software doesn't wear out. But it does deteriorate!

➤ During its life, software will undergo change.

➤ As changes are made, it is likely that errors will be introduced, causing the failure rate curve to spike as shown in the “actual curve” (Figure 1.2).

➤ Before the curve can return to the original steady-state failure rate, another change is requested, causing the curve to spike again. Slowly, the minimum failure rate level begins to rise—the software is deteriorating due to change.



When a hardware component wears out, it is replaced by a spare part.



There are no software spare parts.



Every software failure indicates an error in design or in the process through which design was translated into machine executable code.



Therefore, the software maintenance tasks that accommodate requests for change involve considerably more complexity than hardware maintenance.

### 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.
- Standard screws and off-the-shelf integrated circuits are only two of thousands of standard components that are used by mechanical and electrical engineers as they design new systems.
- 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.




In the hardware world, component reuse is a natural part of the engineering process.



In the software world, it is something that has only begun to be achieved on a broad scale.



A software component should be designed and implemented so that it can be reused in many different programs.



For example, today's interactive user interfaces are built with reusable components that enable the creation of graphics windows, pull-down menus, and a wide variety of interaction mechanisms.



The data structures and processing detail required to build the interface are contained within a library of reusable components for interface construction.

# Software Application Domains

✓ Today, seven broad categories of computer software present continuing challenges for software engineers:

## 1. System software

➤ A collection of system programs known as System Software.

➤ Some system software (e.g., compilers, editors, and file management utilities) processes complex, but determinate, information structures.

➤ Other systems applications (e.g., operating system components, drivers, networking software, telecommunications processors) process largely indeterminate data.

Software is **determinate** if the order and timing of inputs, processing, and outputs is predictable.

Software is ***indeterminate*** if the order and timing of inputs, processing, and outputs cannot be predicted in advance.

## 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.

➤ In addition to conventional data processing applications, application software is used to control business functions in real time.

## 3. Engineering/scientific software

➤ It 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

➤ They reside within a product or system and is used to implement and control features and functions for the end user and for the system itself.

➤ Embedded software can perform limited and esoteric functions (e.g., key pad control for a microwave oven) or provide significant function and control capability (e.g., digital functions in an automobile such as fuel control, dashboard displays, and braking systems).

## 5. Product-line software

➤ They are 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) or address mass consumer markets (e.g., word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, and personal and business financial applications).



## 6. Web Applications

➤ They are also called as “WebApps,” this network-centric software category spans a wide array of applications.

➤ In their simplest form, WebApps can be little more than a set of linked hypertext files that present information using text and limited graphics.

## 7. Artificial intelligence software


➤ It 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.


# Software Engineering

- ❖ 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.
- ❖ **It follows that a concerted effort should be made to understand the problem before a software solution is developed.**
- ✓ The information technology requirements demanded by individuals, businesses,
- ✓ and governments grow increasingly complex with each passing year.
- ✓ Large teams of people now create computer programs.
- ✓ 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.
- ✓ **It follows that design becomes a pivotal activity.**

- 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.
  - **It follows that software should exhibit high quality.**
- 
- 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.
  - **It follows that software should be maintainable.**



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.



Software Engineering is 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 is a layered technology.



Referring to Figure 1.3, any engineering approach (including software engineering) must rest on an organizational commitment to quality.

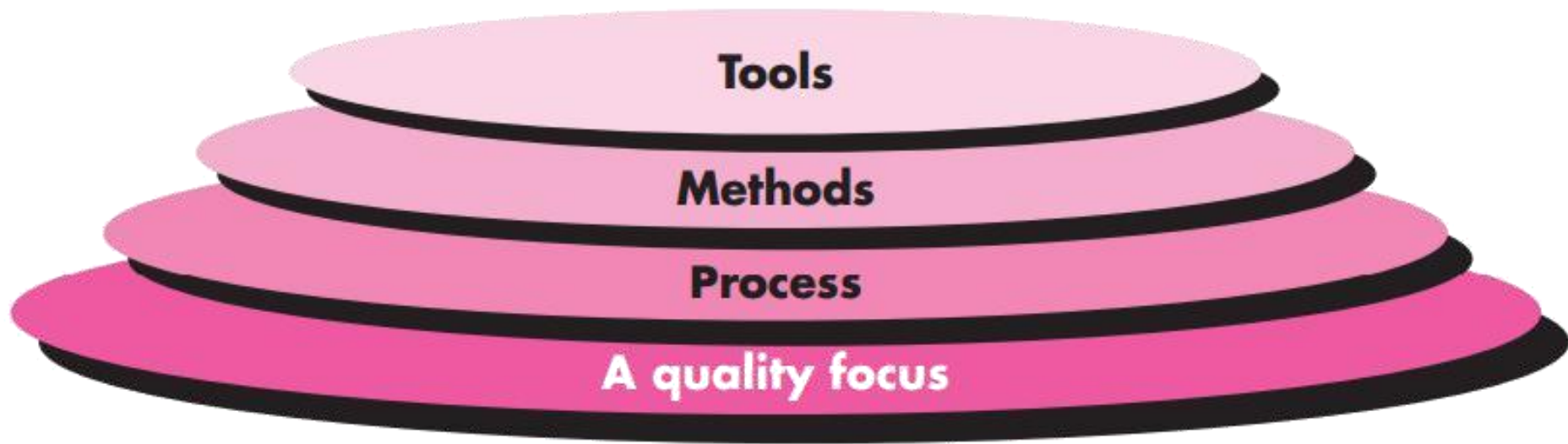


Figure 1.3 :- Software Engineering Layers

➤ 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.

➤ The software process forms the basis for management control of software projects.

➤ 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 modeling, program construction, testing, and support.



Software engineering methods rely on a set of basic principles that govern each area of the technology and include modeling activities and other descriptive techniques.



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



When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called **Computer-aided software engineering**, is established.



# The Software Process

✓ A ***Process*** is a collection of **activities, actions, and tasks** that are performed when some work product is to be created.

✓ 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.

✓ An **action** (e.g., architectural design) encompasses a set of tasks that produce a major work product (e.g., an architectural design model).

✓ A **task** focuses on a small, but well-defined objective (e.g., conducting a unit test) that produces a tangible outcome.

✓ Process is an adaptable approach that enables the people doing the work (the software team) to pick and choose the appropriate set of work actions and tasks.

✓ 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.

✓ In addition, the process framework encompasses a set of **umbrella activities** that are applicable across the entire software process.

✓ A generic **process framework** for software engineering encompasses **five 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 helps to define software features and functions.

# 2. Planning



Any complicated journey can be simplified if a map exists.



A software project is a complicated journey, and the planning activity creates a “map” that helps guide the team as it makes the journey.



The map—called a software project plan.

### 3. Modeling



Whether you're a landscaper, a bridge builder, an aeronautical engineer, a carpenter, or an architect, you work with models every day.



You create a “sketch” of the thing so that you'll understand the big picture—what it will look like architecturally and many other characteristics.



If required, you refine the sketch into greater and greater detail in an effort to better understand the problem and how you're going to solve it.



A software engineer does the same thing by creating 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 (as a complete entity or as a partially completed increment) is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation.



A stakeholder is anyone who has a stake in the successful outcome of the project—business managers, end users, software engineers, support people.



These five generic framework activities can be used during the development of small, simple programs, the creation of large Web applications, and for the engineering of large, complex computer-based systems.



The details of the software process will be quite different in each case, but the framework activities remain the same.



Software engineering process framework activities are complemented by a number of umbrella activities.



In general, umbrella activities are applied throughout a software project and help a software team manage and control progress, quality, change, and risk.



Typical **umbrella activities** include:

# **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; can be used in conjunction with all other framework and umbrella activities

## 6. Software Configuration Management



Manages the effects of change throughout the software process.

## 7. Reusability Managemet



Defines criteria for work product reuse (including software components) 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.

# How does process model differs from one another ?

✓ The software engineering process is not a rigid prescription that must be followed dogmatically by a software team.

✓ Rather, it should be agile and adaptable (to the problem, to the project, to the team, and to the organizational culture).

✓ Therefore, a process adopted for one project might be significantly different than a process adopted for another project.

✓ Among the differences are

➤ Overall flow of activities, actions, and tasks and the interdependencies among them.

➤ Degree to which actions and tasks are defined within each framework activity.

➤ Degree to which work products are identified and required.

➤ Manner in which quality assurance activities are applied.

- ✓ Manner in which project tracking and control activities are applied.
- ✓ Overall degree of detail and rigor with which the process is described.
- ✓ Degree to which the customer and other stakeholders are involved with the project.
- ✓ Level of autonomy given to the software team.
- ✓ Degree to which team organization and roles are prescribed.

# Software Engineering Practice



In the sections that follow, you'll gain a basic understanding of the generic concepts and principles that apply to framework activities

## ***The Essence of Practice***

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

# 1. Understand the problem (communication and analysis).



It's sometimes difficult to admit, but most of us suffer from hubris (presumption) when we're presented with a problem. We listen for a few seconds and then think, Oh yeah, I understand, let's get on with solving this thing.



Unfortunately, understanding isn't always that easy. It's worth spending a little time answering a few simple questions:



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?

## 2. Plan the Solution



Now you understand the problem (or so you think) and you can't wait to begin coding. Before you do, slow down just a bit and do a little design.



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?



### 3. Carry out the Plan



The design you've created serves as a road map for the system you want to build. There may be unexpected detours, and it's possible that you'll discover an even better route as you go, but the "plan" will allow you to proceed without getting lost.



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?

## 4. Examine the Result



You can't be sure that your solution is perfect, but you can be sure that you've designed a sufficient number of tests to uncover as many errors as possible.



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?



In fact, it's reasonable to state that a commonsense approach to software engineering will never lead you astray (away from the right path).

# Software Engineering Principles

✓ David Hooker [Hoo96] has proposed seven principles that focus on software engineering practice as a whole. They are reproduced in the following paragraphs:

## **1. *The First Principle: The Reason It All Exists***

✓ A software system exists for one reason: **to provide value to its users**. All decisions should be made with this in mind.

✓ Before specifying a system requirement, before noting a piece of system functionality, before determining the hardware platforms or development processes, ask yourself questions such as: “Does this add real value to the system?” If the answer is “no,” don’t do it. All other principles support this one.

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

✓ Software design is not a haphazard (carelessness) process. There are many factors to consider in any design effort.

✓ All design should be as simple as possible.

✓ This facilitates having a more easily understood and easily maintained system.

✓ This is not to say that features, even internal features, should be discarded in the name of simplicity.

✓ Indeed, the more elegant designs are usually the more simple ones. Simple also does not mean “quick and dirty.”

### 3. The Third Principle: Maintain the Vision



A clear vision is essential to the success of a software project .



Without conceptual integrity, a system threatens to become a patchwork of incompatible designs, held together by the wrong kind of screws. . . . **Compromising the architectural vision of a software system weakens** and will eventually break even the well-designed systems.



Having an empowered architect who can hold the vision and enforce compliance helps ensure a very successful software project.

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

✓ So, always specify, design, and implement knowing someone else will have to understand what you are doing.

✓ The audience for any product of software development is potentially large.

✓ Specify with an eye to the users. Design, keeping the implementers in mind. Code with concern for those that must maintain and extend the system.

✓ Someone may have to debug the code you write, and that makes them a user of your code. **Making their job easier adds value to the system.**

## 5. The Fifth Principle: Be Open to the Future



A system with a long lifetime has more value. In today's computing environments, where specifications change on a moment's notice and hardware platforms are obsolete just a few months old, software lifetimes are typically measured in months instead of years.



However, true “industrial-strength” software systems must endure far longer.



To do this successfully, these systems must be ready to adapt to these and other changes.



Systems that do this successfully are those that have been designed this way from the start.



## 6. The Sixth Principle: Plan Ahead for Reuse



Reuse saves time and effort. Achieving a high level of reuse is arguably the hardest goal to accomplish in developing a software system.



The reuse of code and designs has been proclaimed as a major benefit of using object-oriented technologies.



However, the return on this investment is not automatic. To leverage (strategic advantage) the reuse possibilities that object-oriented [or conventional] programming provides requires forethought and planning.



There are many techniques to realize reuse at every level of the system development process. . . .



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

## 7. The Seventh principle: Think!

✓ When you think about something, you are more likely to do it right. You also gain knowledge about how to do it right again.

✓ If you do think about something and still do it wrong, it becomes a valuable experience.

✓ A side effect of thinking is learning to recognize when you don't know something, at which point you can research the answer.

✓ When clear thought has gone into a system, value comes out.

✓ Applying the first six principles requires intense thought, for which the potential rewards are enormous.

**( If every software engineer and every software team simply followed Hooker's seven principles, many of the difficulties we experience in building complex computer-based systems would be eliminated. )**

# Software Myths



Software myths—erroneous beliefs about software and the process that is used to build it—can be traced to the earliest days of computing.



Myths have a number of attributes that make them insidious.




For instance, they appear to be reasonable statements of fact (sometimes containing elements of truth), they have an intuitive feel, and they are often promulgated by experienced practitioners who “know the score.”


## 1. Management myths




Managers with software responsibility, like managers in most disciplines, are often under pressure to maintain budgets, keep schedules from slipping, and improve quality.




**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 is it used? Are software practitioners aware of its existence? Does it reflect modern software engineering practice? Is it complete? Is it adaptable? Is it streamlined to improve time-to-delivery while still maintaining a focus on quality? In many cases, the answer to all of these questions is “no.”




**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. 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 out-sources software projects.

## 2. Customer myths




A customer who requests computer software may be a person at the next desk, a technical group down the hall, the marketing/sales department, or an outside company that has requested software under contract.



Myths lead to false expectations (by the customer) and, ultimately, dissatisfaction with the developer.




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



**Reality** : Although a comprehensive and stable statement of requirements is not always possible, an ambiguous “statement of objectives” is a recipe for disaster. Unambiguous requirements (usually derived iteratively) are developed only through effective and continuous communication between customer and developer.



**Myth** : Software 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 requirements changes are requested early (before design or code has been started), the cost impact is relatively small.



### 3. Practitioner's myths

✓ Myths that are still believed by software practitioners have been fostered by over 50 years of programming culture.


✓ During the early days, programming was viewed as an art form. Old ways and attitudes die hard.

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

➤ **Reality** : Someone once said that “the sooner you begin ‘writing code,’ the longer it’ll take you to get done.” 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 technical **review**. Software reviews are a “quality filter” that have been found to be more effective than testing for finding certain classes of software defects.




**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 elements. A variety of work products (e.g., models, documents, plans) provide a foundation for successful engineering and, more important, guidance for software support.



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



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

# Process Models

# A Generic Process Model



When you work to build a product or system, it's important to go through a series of predictable steps—a road map that helps you create a timely, high-quality result.



The road map that you follow is called a “software process”.

## Generic Process Model



A process was defined as a collection of work activities, actions, and tasks that are performed when some work product is to be created.



Each of these activities, actions, and tasks reside within a framework or model that defines their relationship with the process and with one another.

# Software process

## Process framework

### Umbrella activities

#### framework activity # 1

software engineering action #1.1

Task sets

⋮

software engineering action #1.k

Task sets

work tasks  
work products  
quality assurance points  
project mile stones

work tasks  
work products  
quality assurance points  
project mile stones

⋮

#### framework activity # n

software engineering action #n.1

Task sets

⋮

software engineering action #n.m

Task sets

work tasks  
work products  
quality assurance points  
project mile stones

work tasks  
work products  
quality assurance points  
project mile stones

✓ The software process is represented schematically in Figure 2.1.

✓ Referring to the figure, each framework activity is populated by a **set of software engineering actions**.

✓ Each software engineering action is defined by a task set that identifies the work tasks that are to be completed, the work products that will be produced, the quality assurance points that will be required, and the milestones that will be used to indicate progress.

✓ Figure 2 shows Process flow.

✓ A **linear process flow** executes each of the five framework activities in sequence, beginning with communication and culminating with deployment (Figure 2.2a)

✓ An **iterative process flow** repeats one or more of the activities before proceeding to the next (Figure 2.2b).





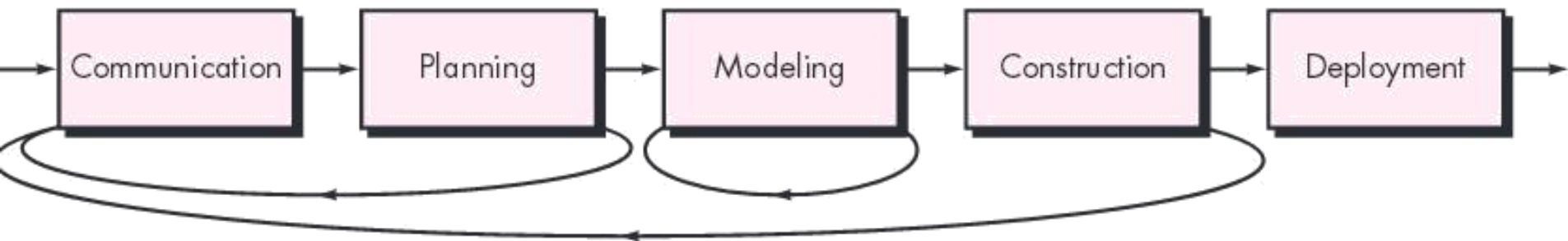
An evolutionary process flow executes the activities in a “circular” manner. Each circuit through the five activities leads to a more complete version of the software (Figure 2.2c).



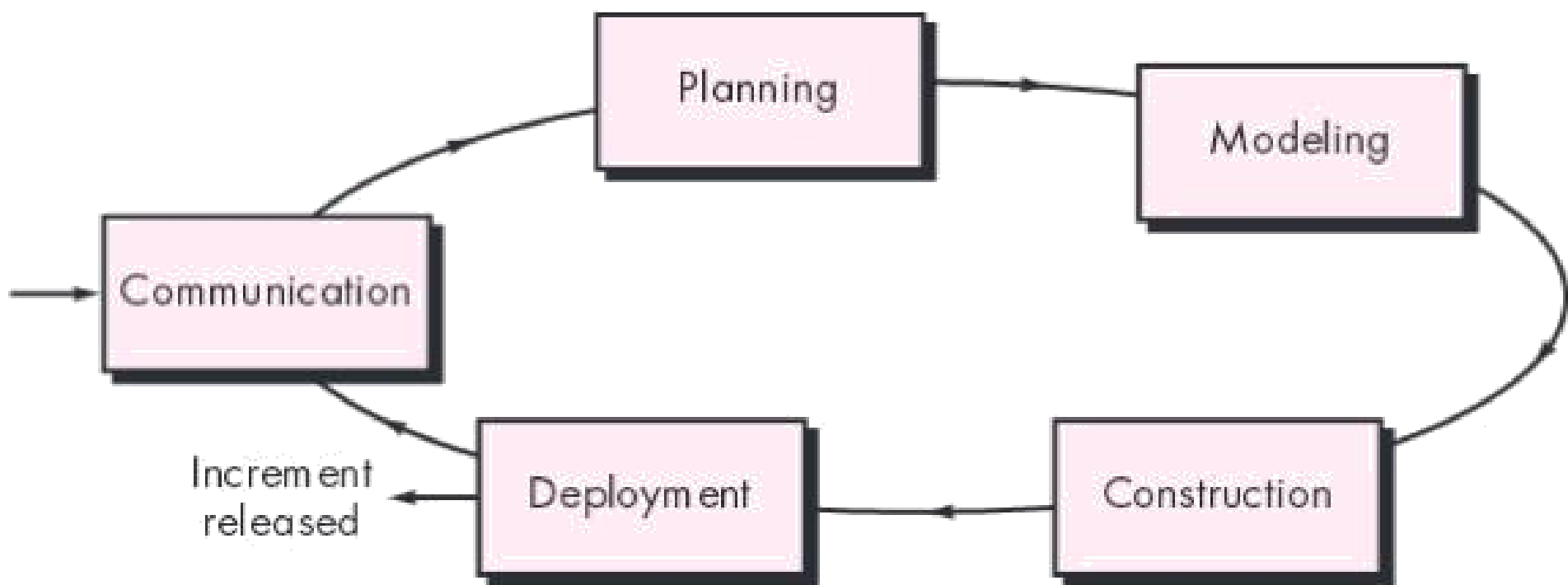
A parallel process flow (Figure 2.2d) executes one or more activities in parallel with other activities (e.g., modeling for one aspect of the software might be executed in parallel with construction of another aspect of the software).



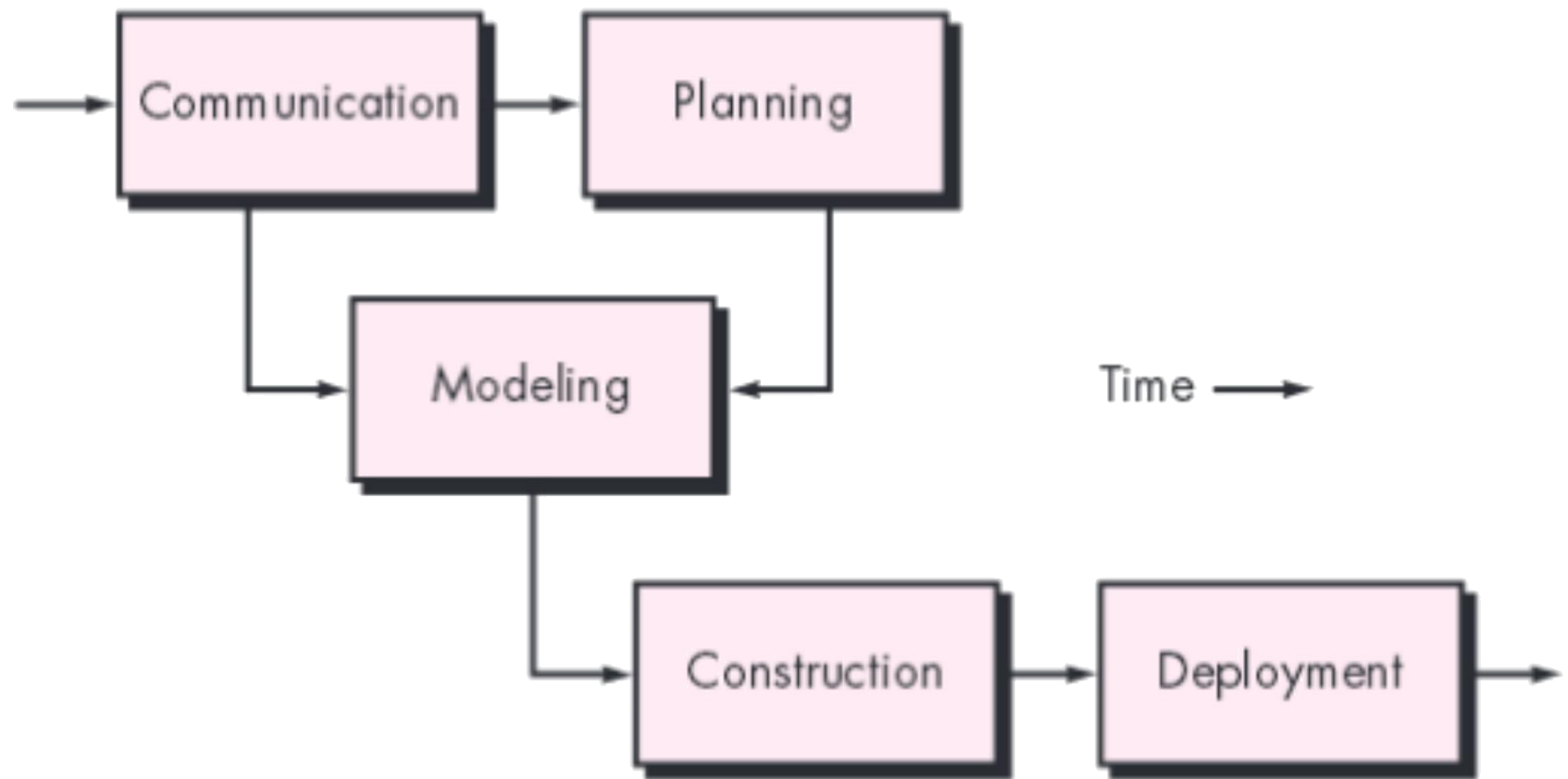
(a) Linear process flow



(b) Iterative process flow



(c) Evolutionary process flow



(d) Parallel process flow

**Figure 2.2 : Process Flow**

# Prescriptive Process Models:



We call them 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.

# 1. The Waterfall Model



There are times when the requirements for a problem are well understood—when work flows from communication through deployment in a reasonably linear fashion.

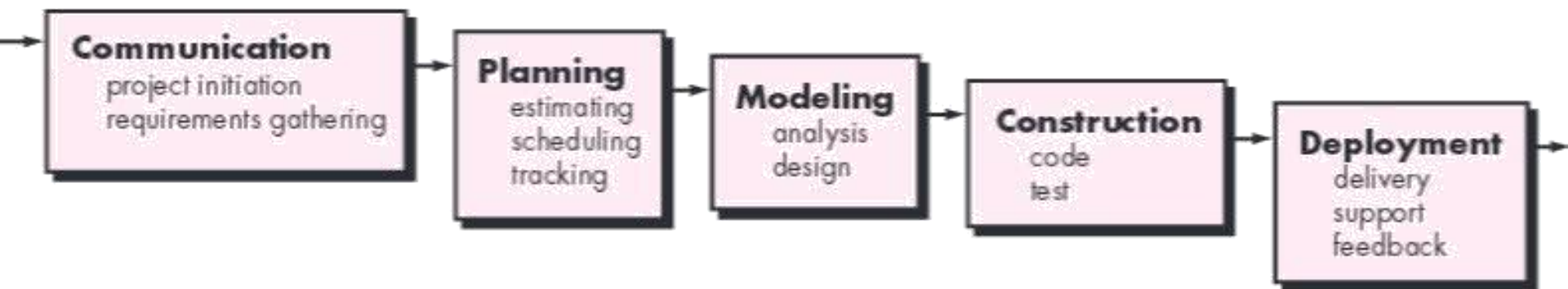


The **waterfall model**, sometimes called the **classic life cycle**, suggests a systematic, sequential approach to software development that begins with customer specification of requirements and progresses through planning, modeling, construction, and deployment, culminating in ongoing support of the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

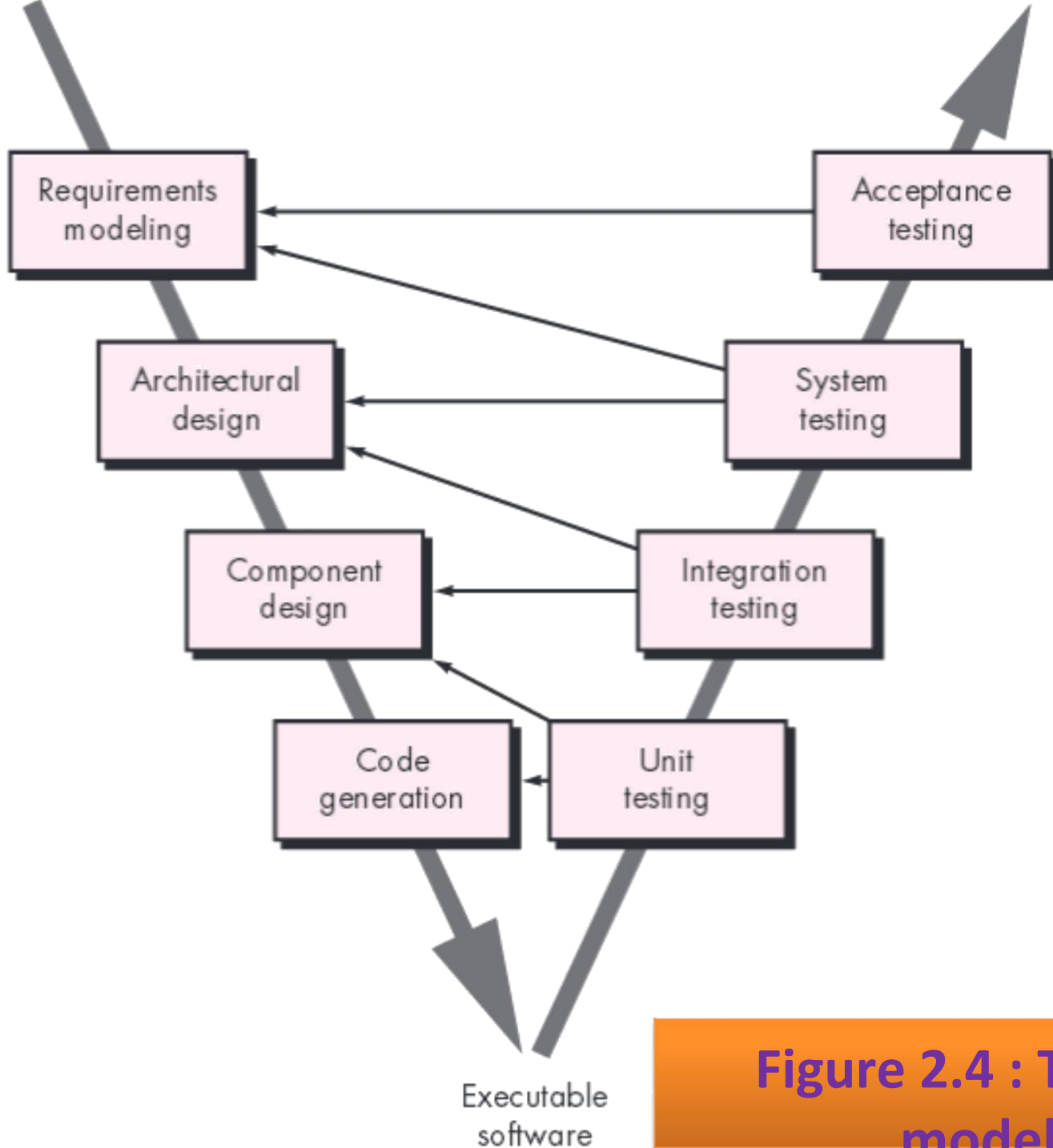


A variation in the representation of the waterfall model is called the **V-model**. Represented in Figure 2.4.





**Figure 2.3 : The Waterfall Model**



**Figure 2.4 : The V-model**



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.



In reality, there is no fundamental difference between the classic life cycle and the V-model.



The V-model provides a way of visualizing how verification and validation actions are applied to earlier engineering work.



The waterfall model is the oldest paradigm for software engineering.



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

1. 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.
2. 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.
3. The customer must have patience. A working version of the program(s) will not be available until late in the project time span. A major blunder, if undetected until the working program is reviewed, can be disastrous.

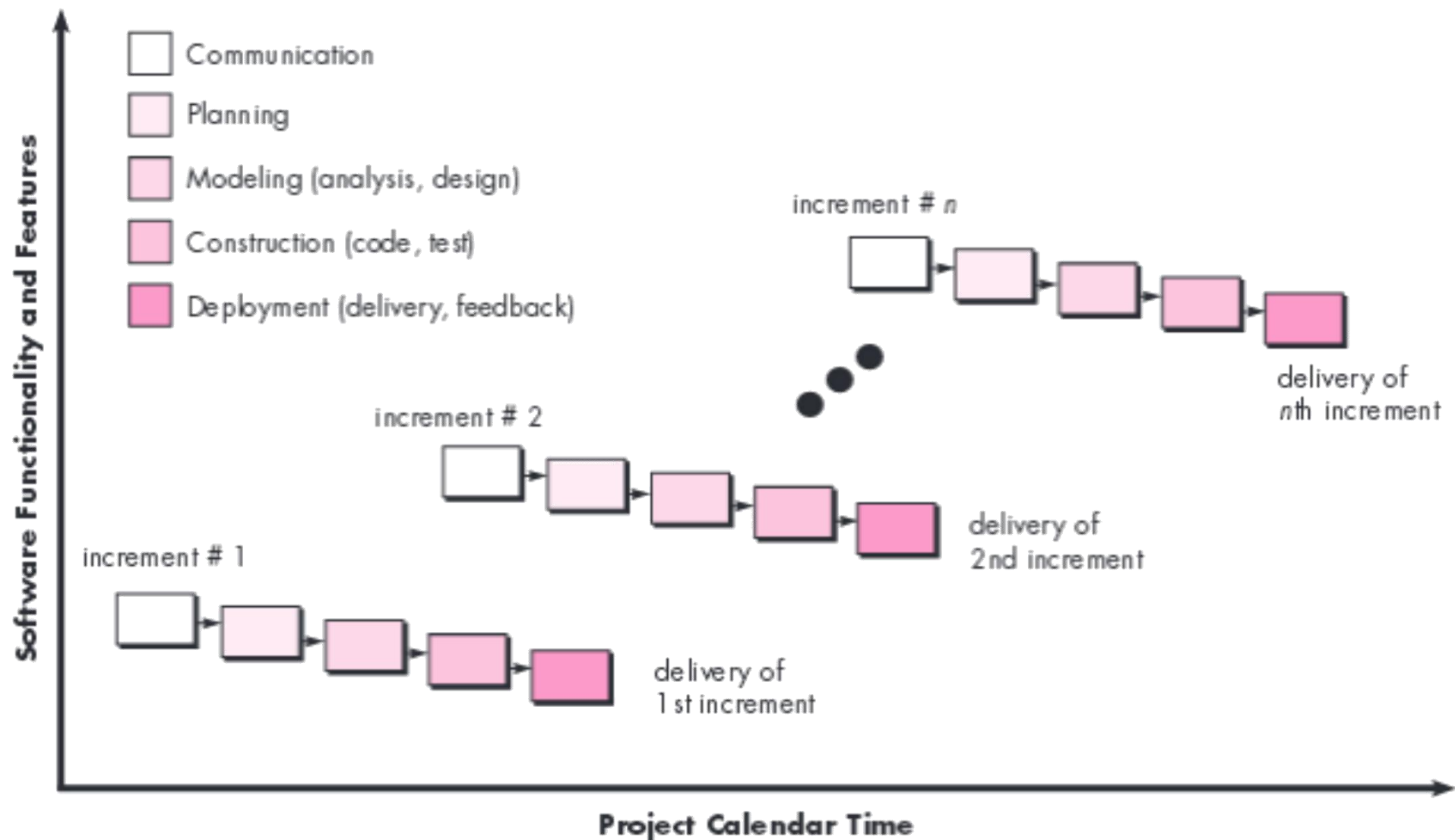


The linear nature of the classic life cycle leads to “blocking states” in which some project team members must wait for other members of the team to complete dependent tasks.



In fact, the time spent waiting can exceed the time spent on productive work.

## 2. Incremental Process Models (RAD)



**Figure 2.4 : The Incremental Model**

✓ The incremental model combines elements of linear and parallel process flow.

✓ **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.

✓ It should be noted that the process flow for any increment can incorporate the prototyping paradigm.

✓ When an incremental model is used, the first increment is often a core product. That is, basic requirements are addressed but many supplementary features (some known, others unknown) remain undelivered.





The core product is used by the customer (or undergoes detailed evaluation).



As a result of use and/or evaluation, 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.



The incremental process model focuses on the delivery of an operational product with each increment.



Early increments are stripped-down versions of the final product, but they do provide capability that serves the user and also provide a platform for evaluation by the user.



Early increments can be implemented with fewer people. If the core product is well received, then additional staff (if required) can be added to implement the next increment. In addition, increments can be planned to manage technical risks.



For example, a major system might require the availability of new hardware that is under development and whose delivery date is uncertain.



It might be possible to plan early increments in a way that avoids the use of this hardware, thereby enabling partial functionality to be delivered to end users without inordinate delay.

# 3. Evolutionary Process

- ✓ Software, like all complex systems, evolves over a period of time.
- ✓ Business and product requirements often change as development proceeds.
- ✓ You need a process model that has been explicitly designed to accommodate a product that evolves over time.

## Prototyping

- ✓ Often, a customer defines a set of general objectives for software, but does not identify detailed requirements for functions and features.
- ✓ In other cases, the developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system, or the form that human machine interaction should take.
- ✓ In these, and many other situations, a prototyping paradigm may offer the best approach.

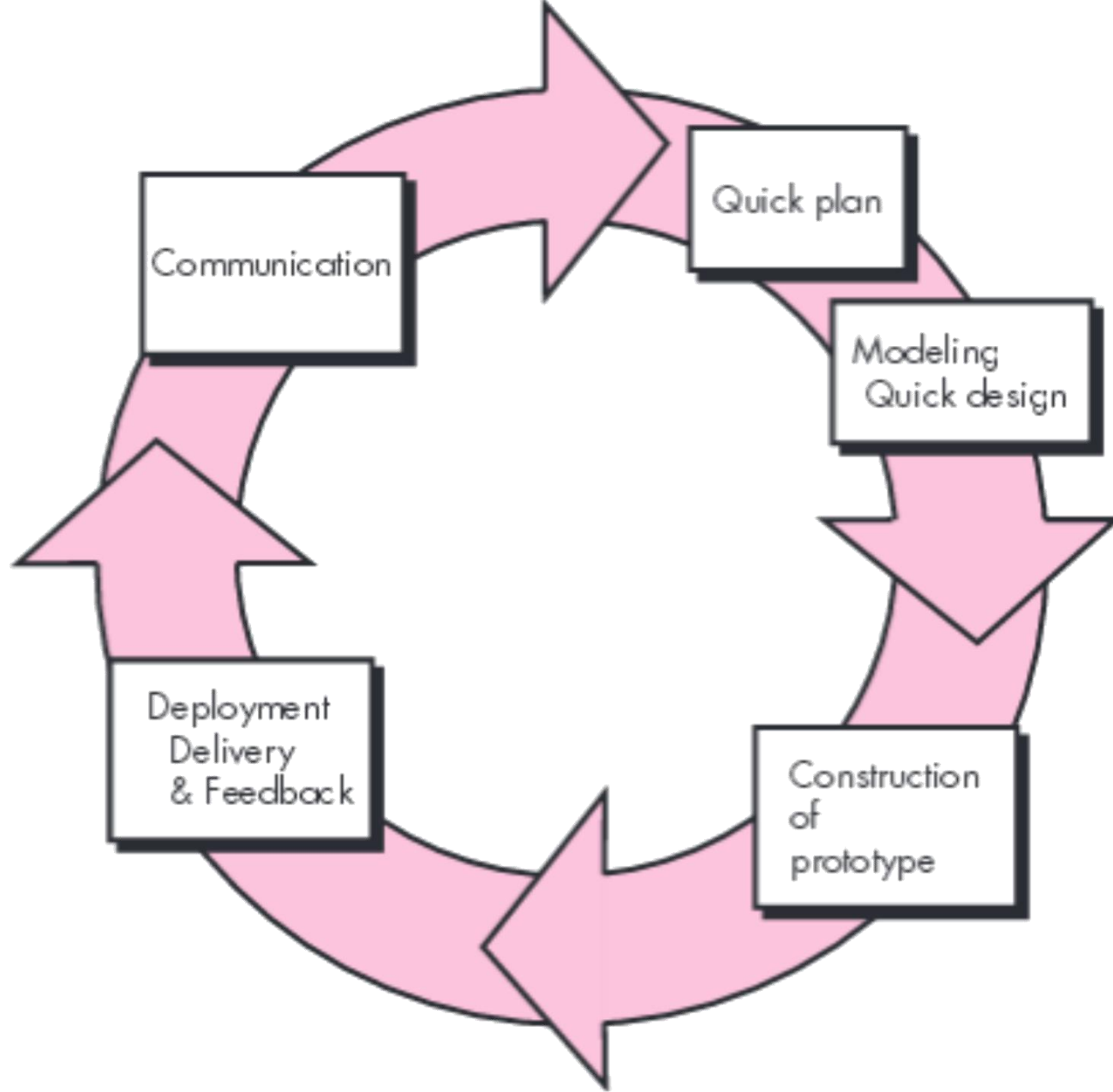
✓ The prototyping paradigm (Figure 2.6) begins with communication.

✓ You meet with other stakeholders to define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory.

✓ A prototyping iteration is planned quickly, and modeling (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 user.

✓ 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.



**Figure 2.6 : The Prototyping Paradigm**

✓ Ideally, the prototype serves as a mechanism for identifying software requirements.

✓ If a working prototype is to be built, you can make use of existing program fragments or apply tools (e.g., report generators and window managers) that enable working programs to be generated quickly.

✓ Both stakeholders and software engineers like the prototyping paradigm.

✓ Users get a feel for the actual system, and developers get to build something immediately.

✓ **Yet, prototyping can be problematic for the following reasons:**

1. Stakeholders see what appears to be a working version of the software, unaware that the prototype is held together haphazardly, unaware that in the rush to get it working you haven't considered overall software quality or long-term maintainability.
2. As a software engineer, you often make implementation compromises in order to get a prototype working quickly.



## 4. Unified Process



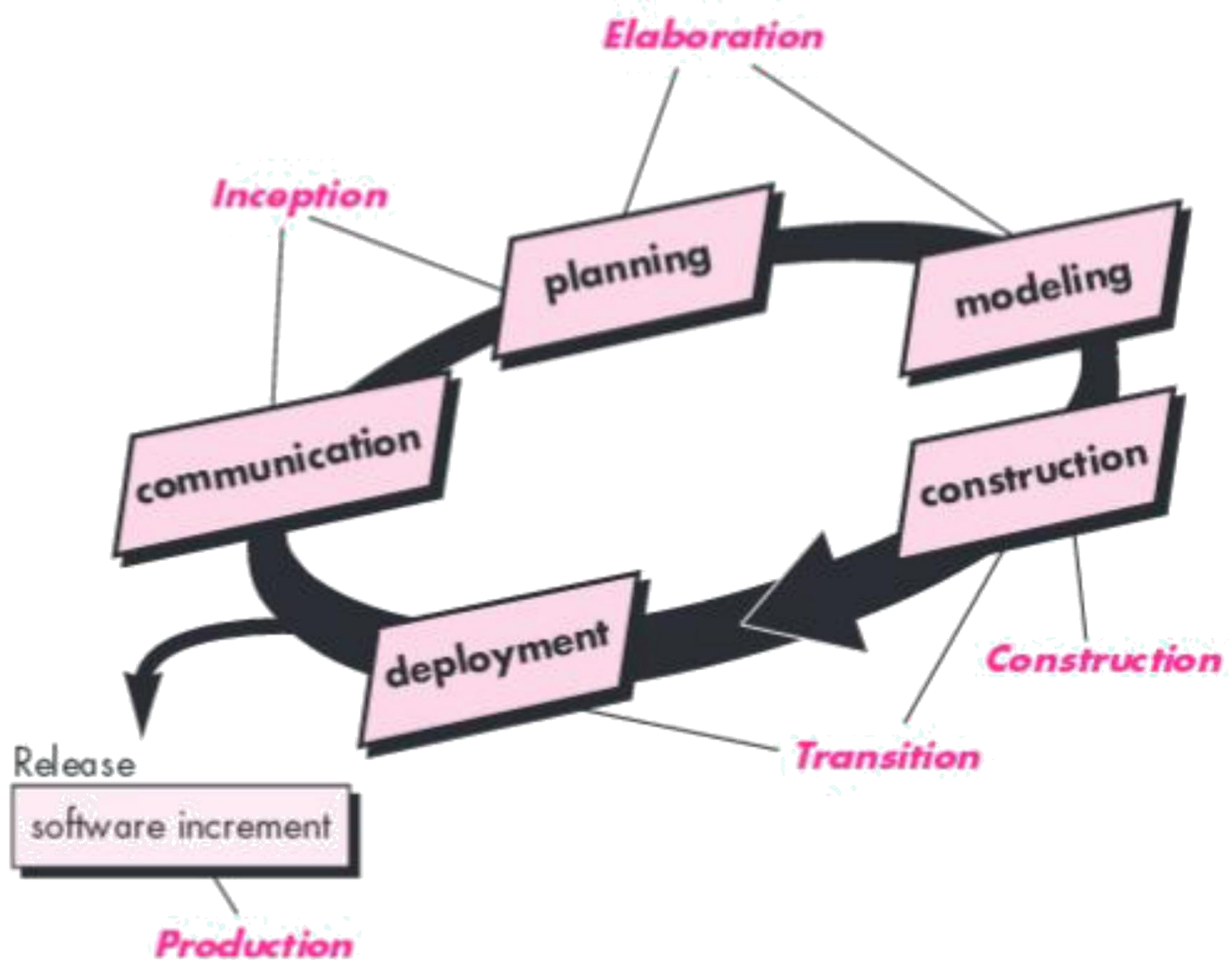
The Unified Process recognizes the importance of customer communication and streamlined methods for describing the customer's view of a system.



“Unified method” that would combine the best features of each of their individual object-oriented analysis and design methods.



The result was UML—a unified modeling language that contains a robust notation for the modeling and development of object-oriented systems.



**Figure 2.9 : The Unified Process**

# Phases of the Unified Process

✓ The **Inception phase** of the UP encompasses both customer communication and planning activities.

✓ By collaborating with stakeholders, business requirements for the software are identified; a rough architecture for the system is proposed; and a plan for the iterative, incremental nature of the ensuing project is developed.

✓ Architecture at this point is nothing more than a tentative outline of major subsystems and the function and features that populate them. Later, the architecture will be refined and expanded into a set of models that will represent different views of the system.

✓ The **Elaboration phase** encompasses the communication and modeling activities of the generic process model (Figure 2.9).

✓ Elaboration refines and expands the preliminary use cases that were developed as part of the inception phase and expands the architectural representation.

✓ The architectural baseline demonstrates the viability of the architecture but does not provide all features and functions required to use the system.

✓ The **Construction phase** of the UP is identical to the construction activity defined for the generic software process.

✓ Using the architectural model as input, the construction phase develops or acquires the software components that will make each use case operational for end users.

✓ To accomplish this, requirements and design models that were started during the elaboration phase are completed to reflect the final version of the software increment.

✓ The **Transition phase** of the UP encompasses the latter stages of the generic construction activity and the first part of the generic deployment (delivery and feedback) activity.

✓ Software is given to end users for beta testing and user feedback reports both defects and necessary changes.

✓ In addition, the software team creates the necessary support information (e.g., user manuals, troubleshooting guides, installation procedures) that is required for the release.

✓ The **Production phase** of the UP coincides with the deployment activity of the generic process.

✓ During this phase, the ongoing use of the software is monitored, support for the operating environment (infrastructure) is provided, and defect reports and requests for changes are submitted and evaluated.

# 5. Concurrent Model

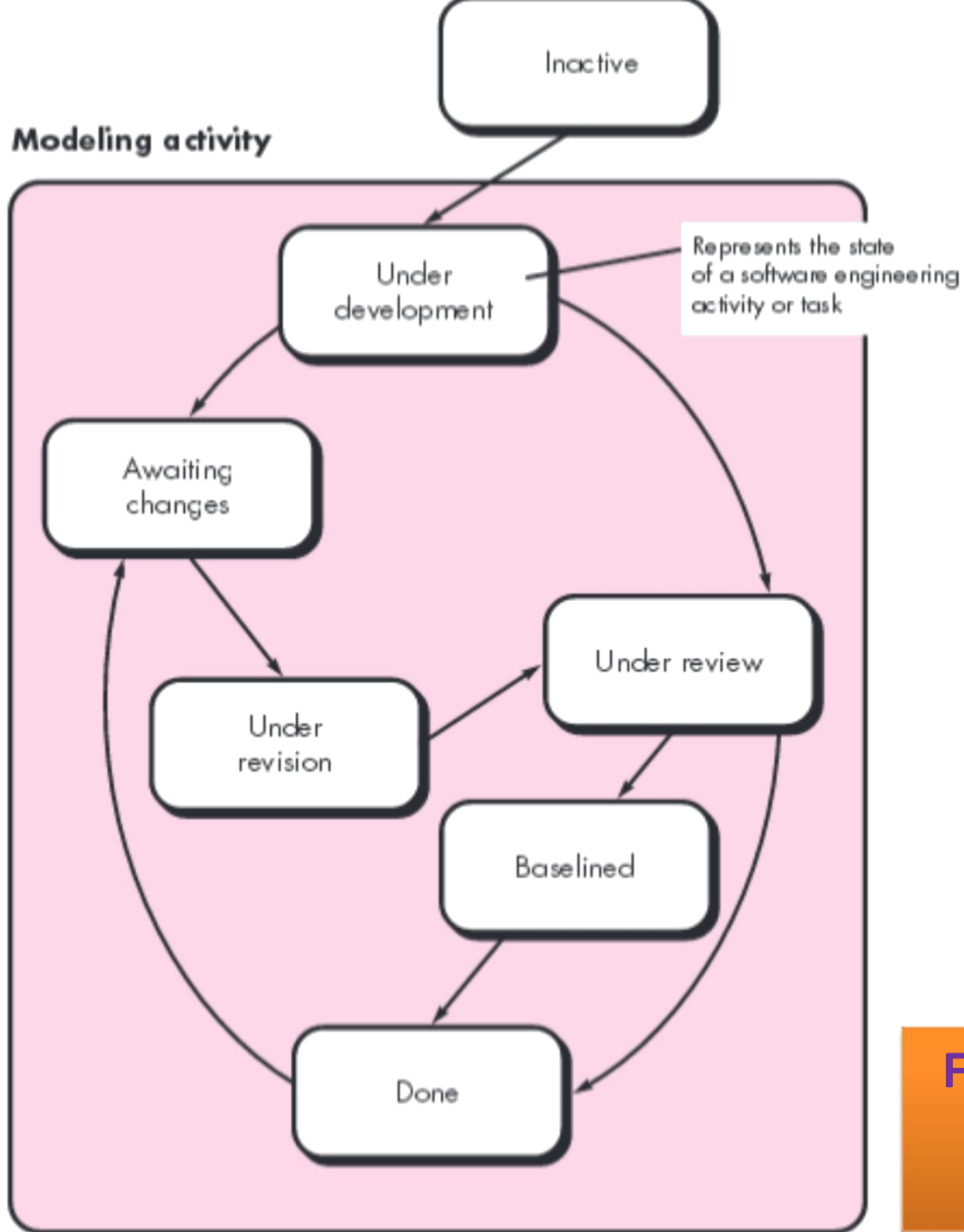
✓ The **concurrent development model**, sometimes called **concurrent engineering**, allows a software team to represent iterative and concurrent elements of any of the process models.

✓ Figure 2.8 provides a schematic representation of one software engineering activity within the modeling activity using a concurrent modeling approach.

✓ The activity **modeling** may be in any one of the states noted at any given time.

✓ Software engineering activities exist concurrently but reside in different states.





**Figure 2.8 : One element of the Concurrent process model**



For example, early in a project the communication activity (not shown in the figure) has completed its first iteration and exists in the **awaiting changes** state.



The modeling activity (which existed in the **inactive state** while initial communication was completed, now makes a transition into the **under development** state.



If, however, the customer indicates that changes in requirements must be made, the modeling activity moves from the under **development state** into the **awaiting changes** state.



Concurrent modeling defines a series of events that will trigger transitions from state to state for each of the software engineering activities, actions, or tasks.



This generates the event analysis model correction, which will trigger the requirements analysis action from the **done** state into the **awaiting changes** state.

# Advanced Process Models & Tools: Agile software development:

# Agile Methods

## What is it ?

✓ Agile software engineering combines a philosophy and a set of development guidelines.

✓ The philosophy encourages customer satisfaction and early incremental delivery of software; small, highly motivated project teams; informal methods; minimal software engineering work products; and overall development simplicity.

✓ An agile team fosters communication and collaboration among all who serve on it.

## Why is it important ?

✓ The modern business environment that spawns computer-based systems and software products is fast-paced and ever changing.

# What is an Agile Process ?



Any agile software process is characterized in a manner that addresses a number of key assumptions about the majority of software projects.

1. It is difficult to predict in advance which software requirements will persist and which will change. It is equally difficult to predict how customer priorities will change as the project proceeds.
2. For many types of software, design and construction are interleaved. That is, both activities should be performed in tandem so that design models are proven as they are created. It is difficult to predict how much design is necessary before construction is used to prove the design.
3. Analysis, design, construction, and testing are not as predictable (from a planning point of view) as we might like.

✓ When this heavyweight, plan-driven development approach is applied to small and medium-sized business systems, the overhead involved is so large that it dominates the software development process.

✓ More time is spent on how the system should be developed than on program development and testing.

✓ As the system requirements change, rework is essential and, in principle at least, the specification and design has to change with the program.

✓ Dissatisfaction with these heavyweight approaches to software engineering led a number of software developers in the 1990s to propose new '**Agile Methods**'.

✓ These allowed the development team to focus on the software itself rather than on its design and documentation.



Agile methods universally rely on an incremental approach to software specification, development, and delivery.



They are best suited to application development where the system requirements usually change rapidly during the development process.



They are intended to deliver working software quickly to customers, who can then propose new and changed requirements to be included in later iterations of the system.



Although these agile methods are all based around the notion of incremental development and delivery, they propose different processes to achieve this.





Two of the most widely used methods: **Extreme programming & Scrum.**



Agile methods have been very successful for some types of system development:

1. Product development where a software company is developing a small or medium-sized product for sale.
2. Custom system development within an organization, where there is a clear commitment from the customer to become involved in the development process and where there are not a lot of external rules and regulations that affect the software.

## Principles of Agile Methods :-

1. **Customer Involvement** :- Customers should be closely involved throughout the development process. Their role is provide and prioritize new system requirements and to evaluate the iterations of the system.
2. **Incremental delivery** :- The software is developed in increments with the customer specifying the requirements to be included in each increment.
3. **People not process** :- The skills of the development team should be recognized and exploited. Team members should be left to develop their own ways of working without prescriptive processes.
4. **Embrace change** :- Expect the system requirements to change and so design the system to accommodate these changes.
5. **Maintain simplicity** :- Focus on simplicity in both the software being developed and in the development process. Wherever possible, actively work to eliminate complexity from the system.



In practice, the principles underlying agile methods are sometimes difficult to realize:

1. Although the idea of customer involvement in the development process is an attractive one, its success depends on having a customer who is willing and able to spend time with the development team and who can represent all system stakeholders. Frequently, the customer representatives are subject to other pressures and cannot take full part in the software development.
2. Individual team members may not have suitable personalities for the intense involvement that is typical of agile methods, and therefore not interact well with other team members.
3. Prioritizing changes can be extremely difficult, especially in systems for which there are many stakeholders. Typically, each stakeholder gives different priorities to different changes.

4. Maintaining simplicity requires extra work. Under pressure from delivery schedules, the team members may not have time to carry out desirable system simplifications.
5. Many organizations, especially large companies, have spent years changing their culture so that processes are defined and followed. It is difficult for them to move to a working model in which processes are informal and defined by development teams.

# Plan-driven and agile development

✓ By contrast, a plan-driven approach to software engineering identifies separate stages in the software process with outputs associated with each stage.

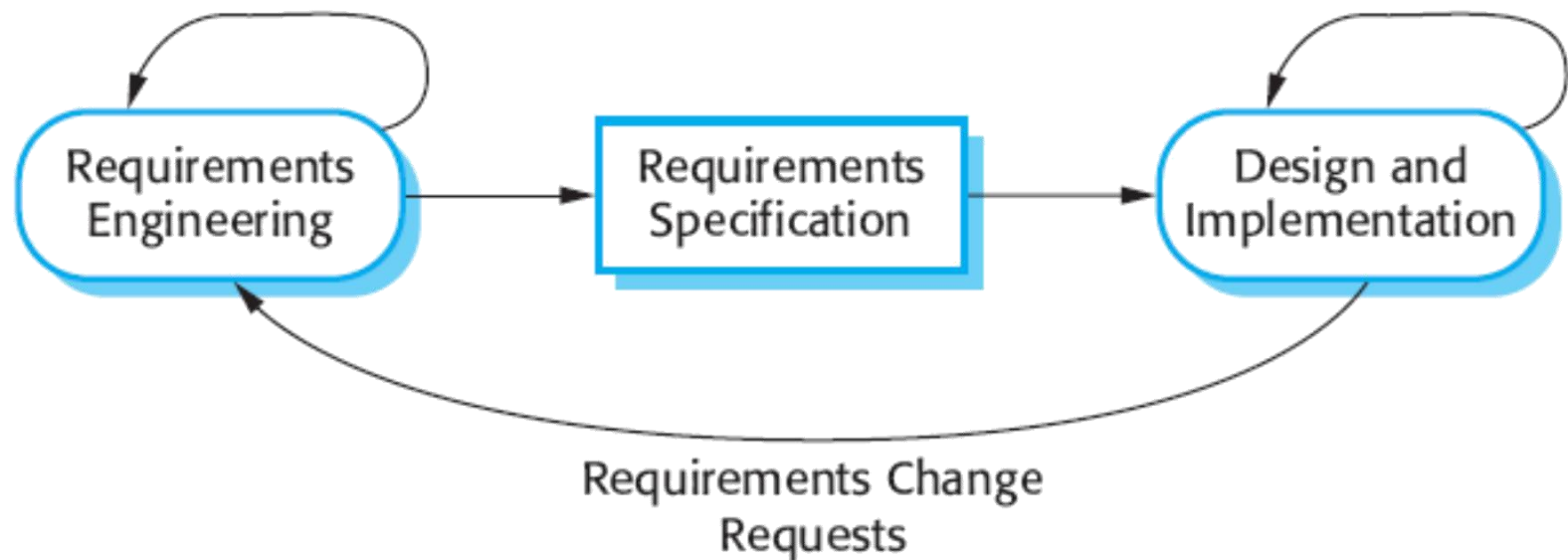
✓ The outputs from one stage are used as a basis for planning the following process activity.

✓ Figure 3.2 shows the distinctions between plan-driven and agile approaches to system specification.

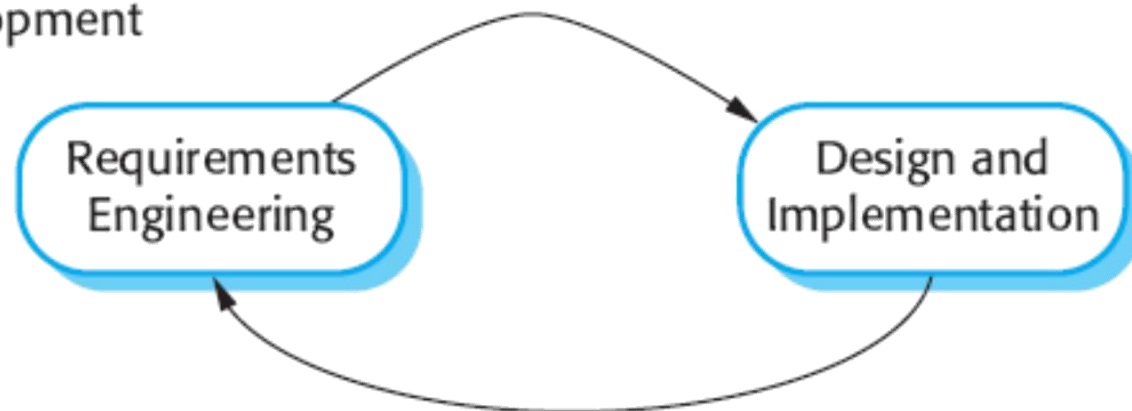
✓ Requirement Engineering = What your program should do ?

✓ Requirement Specification = How you plan to do it ?

## Plan-Based Development



## Agile Development



**Figure 3.2 : Plan-driven and agile specification**

✓ In a plan-driven approach, iteration occurs within activities with formal documents used to communicate between stages of the process.

✓ For example, the requirements will evolve and, ultimately, a requirements specification will be produced.

✓ This is then an input to the design and implementation process. In an agile approach, iteration occurs across activities.

✓ Therefore, the requirements and the design are developed together, rather than separately.

✓ **A plan-driven software process can support incremental development and delivery.**

✓ It is perfectly feasible to allocate requirements and plan the design and development phase as a series of increments.



✓ An agile process is not inevitably code-focused and it may produce some design documentation.

✓ To decide on the balance between a plan-based and an agile approach, you have to answer a range of technical, human, and organizational questions:

1. Is it important to have a very detailed specification and design before moving to implementation? If so, you probably need to use a plan-driven approach.
2. Is an incremental delivery strategy, where you deliver the software to customers and get rapid feedback from them, realistic? If so, consider using agile methods.
3. How large is the system that is being developed? Agile methods are most effective when the system can be developed with a small co-located team who can communicate informally. This may not be possible for large systems that require larger development teams so a plan-driven approach may have to be used.

4. What type of system is being developed? Systems that require a lot of analysis before implementation (e.g., real-time system with complex timing requirements) usually need a fairly detailed design to carry out this analysis. A plan-driven approach may be best in those circumstances.
4. What is the expected system lifetime? Long-lifetime systems may require more design documentation to communicate the original intentions of the system developers to the support team. However, supporters of agile methods rightly argue that documentation is frequently not kept up to date and it is not of much use for long-term system maintenance.

# Extreme programming

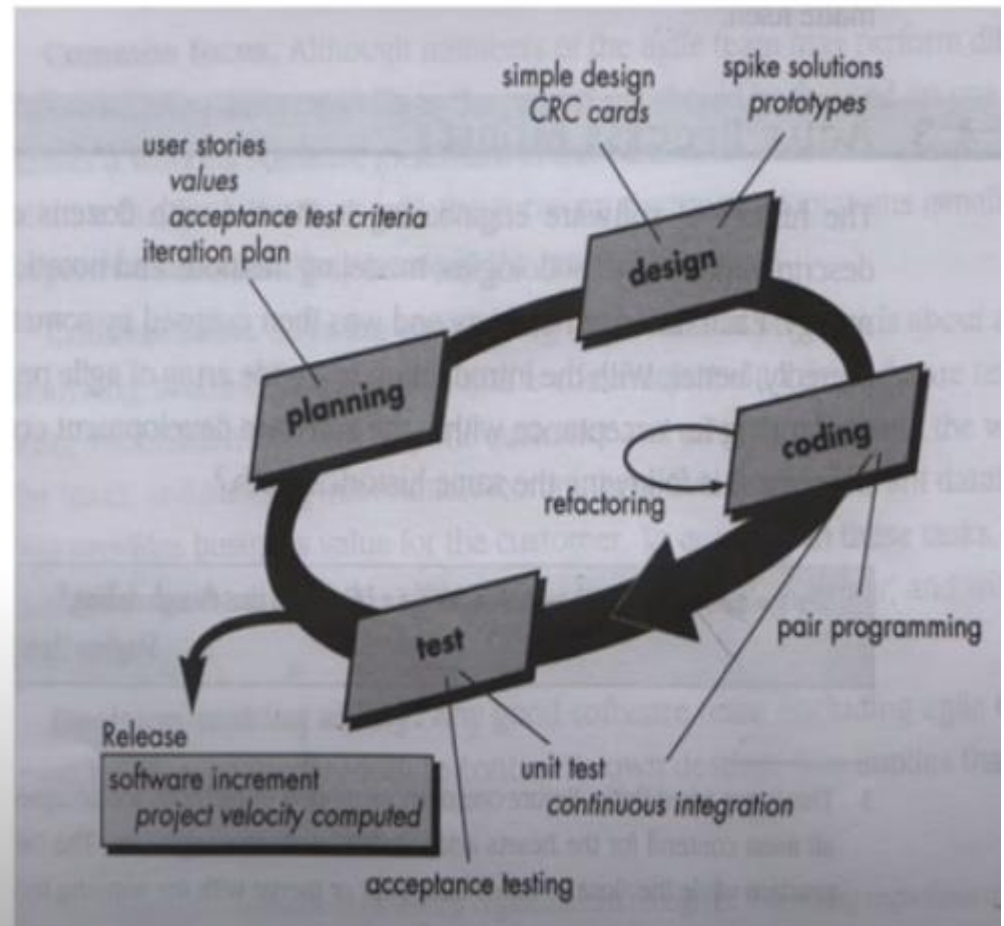
# Extreme programming

- **EXTREME PROGRAMMING(XP)**

- XP is a lightweight , efficient, low-risk, flexible, predictable, scientific, and fun way to develop a software.
- Small to medium sized teams that work under vague and rapidly changing requirements.
- The five values of XP are communication, simplicity, feedback, courage, and respect
- Follows Object Oriented approach.

**Spike solution:** A **spike solution** is a very simple program to explore potential **solutions**. Difficult designs should be modeled using prototype.

**Refactoring:** Improves the internal structure



# Class Responsibility Collaborator(CRC)

- Class-Responsibility Collaborator(CRC).

Class: <i>Librarian</i>	
Responsibilities	Collaborators
<i>check in book</i>	<i>Book</i>
<i>check out book</i>	<i>Book, Borrower</i>
<i>search for book</i>	<i>Book</i>
<i>knows all books</i>	
<i>search for borrower</i>	<i>Borrower</i>
<i>knows all borrowers</i>	

Class CardReader	
Responsibilities	Collaborators
Tell ATM when card is inserted	ATM
Read information from card	Card
Eject card	
Retain card	

**ACCEPTANCE TESTING:-**During the process of manufacturing a ballpoint pen, the cap, the body, the tail and clip, the ink cartridge and the ballpoint are produced separately and unit tested separately. When two or more units are ready, they are assembled and Integration Testing is performed. When the complete pen is integrated, System Testing is performed. Once System Testing is complete, Acceptance Testing is performed so as to confirm that the ballpoint pen is ready to be made available to the end-users.

# Advantages and Disadvantages of XP Programming

## • **ADVANTAGES:-**

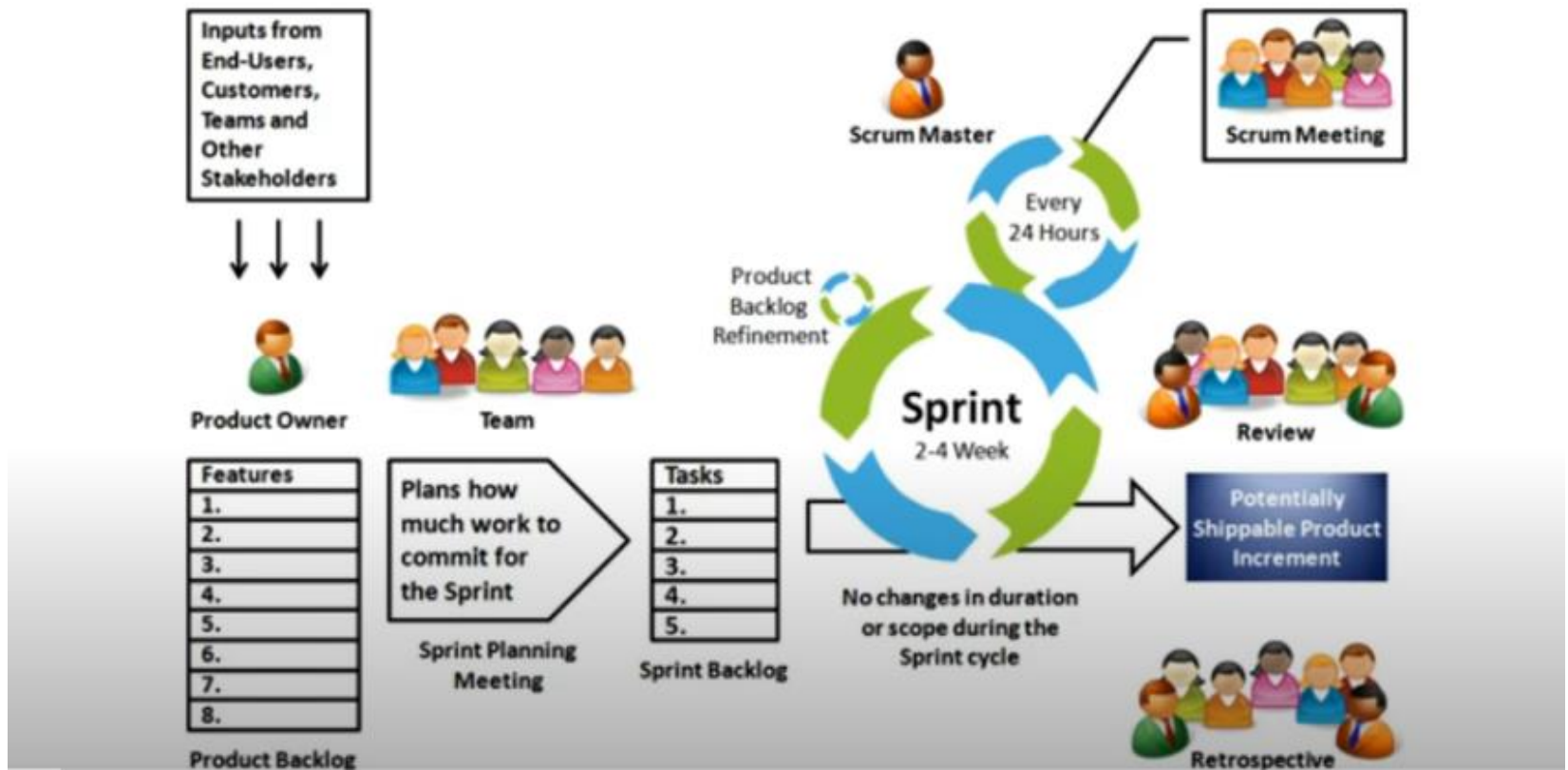
1. Fewer documentation required.
2. Collaboration with customers.
3. Flexibility to developers.
4. Easy to manage.

## • **DISADVANTAGES:-**

1. Depends heavily on customer interaction.
2. Transfer of technology to new team members may be quite challenging due to lack of documentation.

# SCRUM

# SCRUM



# SCRUM Roles

ROLES:- 1.Product Owner.

2.Scrum Master.

3.Scrum Team.

- The product owner creates a product backlog (essentially, a wishlist of tasks that need to be prioritized in a project).
- Decides Release date and content.
- Prioritize features according to market value.
- Accept or Reject work results.



# SCRUM Master and TEAM

- The Scrum Master conducts daily meetings.
  - Ensures that the team is fully functional and productive.
  - Shield the team from external interfaces.
  - Responsible for management of the project.
- 
- Typically 5 to 10 people.
  - Teams are cross functional and self-organizing.
  - Scrum team determines which items and in what order are to be executed.

# Advantages and Disadvantages of SCRUM

## • **ADVANTAGES:-**

- Large projects are broken down into small sprints.
- Customers are involved, so best results are ensured.
- Budget Friendly and Time Saving.
- The team gets clear visibility through scrum meetings.
- Developments are coded and tested during the sprint review

## • **DISADVANTAGES:-**

- Once decided the sprint backlog items cannot be changed.
- Requires strong commitment from all members of the team.
- If any team member leaves in the middle of a project, it can have a huge negative impact on the project
- Quality is hard to implement, until the team goes through aggressive testing process

# Testing in XP

## **1) Test First Development**

- Test Cases

## **2) Incremental Test Development from Scenarios**

- One or more unit tests can be prepared for each task

## **3) Involving User in Testing**

- Acceptance Testing

## **4) Use of Automated Test Tools**

- Junit

# Agile Tools

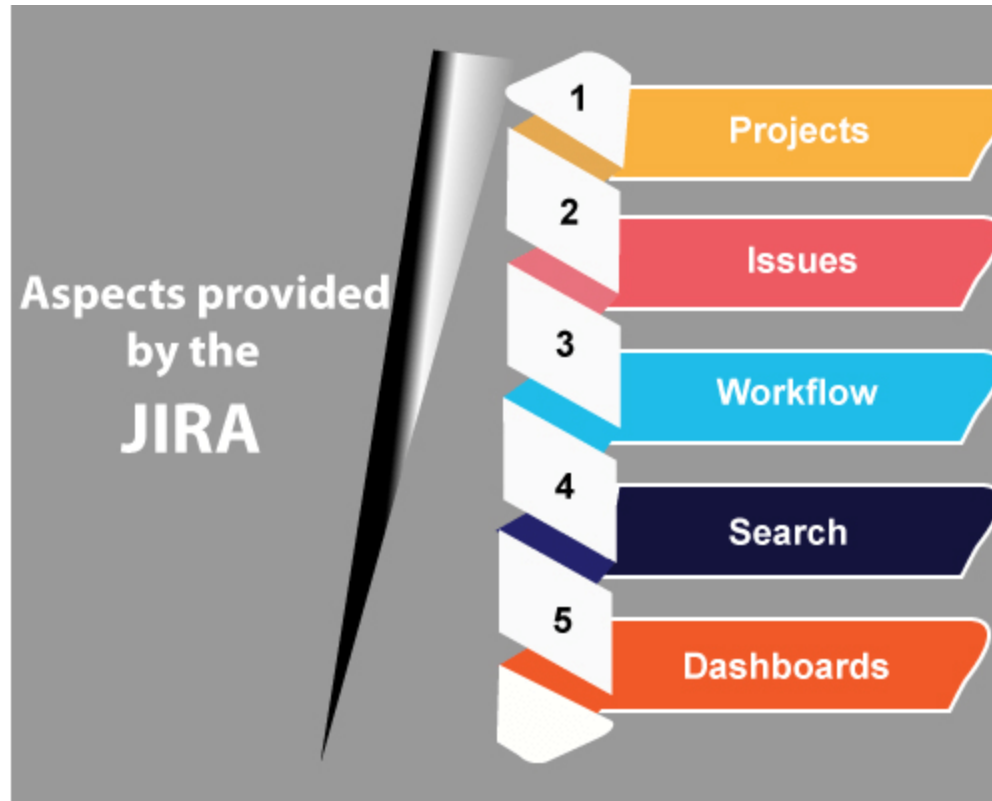
1. JIRA
2. KANBAN

# JIRA

- **What is JIRA?**
  - ✓ JIRA is a software testing tool developed by the Australian Company Atlassian.
  - ✓ It is a bug tracking tool that reports all the issues related to your software or mobile apps.
  - ✓ The word JIRA comes from the Japanese word, i.e., "Gojira" which means Godzilla.
  - ✓ JIRA is based on the Agile methodology and the current version of the Jira is 6.
  - ✓ JIRA is Written in JAVA
  - ✓ JIRA is one of the most widely used open source testing tool used in manual testing.

# JIRA

- The following are the useful aspects provided by the JIRA:



# JIRA

- **Projects:** It is used to manage the defects very effectively.
- **Issue:** It is used to track and manage the defects/issues.
- **Workflow:** Processes the Issue/Defect life cycle. Suppose we have a business requirement, we create the technical design and from the technical design, we create the test cases. After creating the test cases, coding is done, and then testing is performed on the project. This design workflow is possible by using Jira.
- **Search:** Find with ease. Suppose we have done with a project at the beginning of the December and its version is 1.0. Now, we move to version 1.1 and completed at the end of December. What we are doing is that we are adding new versions. Through Jira, we can get to know that what happened in the earlier versions, how many defects occurred in the earlier projects and the learning we achieve from the earlier projects.
- **Dashboards:** Dashboard is a display which you see when you log in to the Jira. You can create multiple dashboards for multiple projects. You can create the personal dashboard and can add the gadgets in a dashboard so that you can keep track of the assignments and issues that you are working on.

# KANBAN

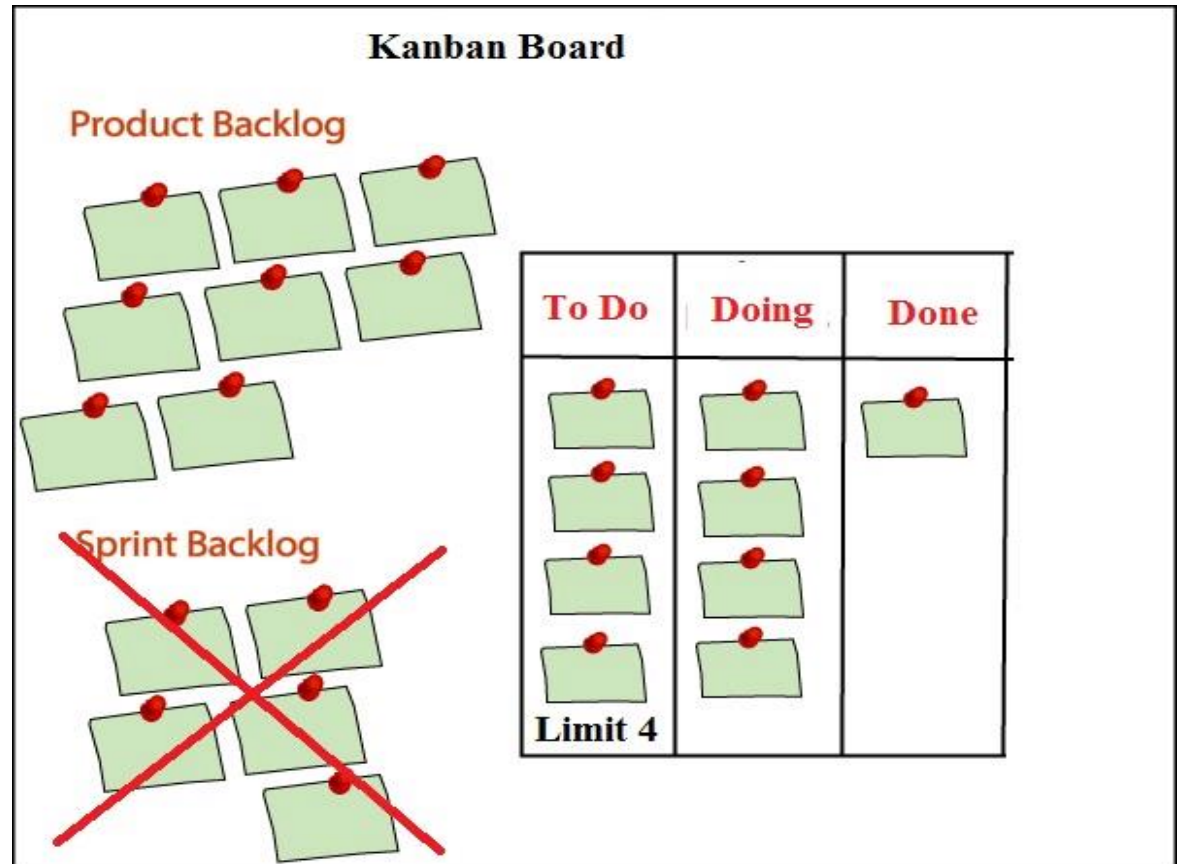
- Kanban is the most popular agile framework after Scrum for software development.
- It provides the real-time and transparency of work.
- In Kanban board, all the tasks are visible that allows the team members to see the state of every task at any time.



# KANBAN

- Kanban Board

Kanban board is a tool used to visualize the work and limit work-in-progress.



# KANBAN

- Kanban board consists of three states:
  - To Do
  - Doing
  - Done
- **TO DO**-When the project is started, then we put all the activities from the product backlog to the 'To Do' state.
- **Doing**-When the team member starts working on an activity, then that activity is put in a 'Doing' state,
- **Done**-when the activity is placed, then it is placed in a 'Done' state.
- From the Kanban board, one can get to know which activities have been done and which activities they need to develop.

# KANBAN

- One of the most important features of the Kanban board is a Limit option.
- In the above figure, we have eight tasks in a product backlog and limit set is 4.
- At a time, it will take only four tasks in a 'To Do' state, and if any of the tasks come in a 'Doing' state, then one more task from the product backlog will be placed in a 'To Do' state.
- In this way, we can set the limit depending on the availability of the resources.

# KANBAN vs SCRUM

- As in scrum, we are taking some activities from a product backlog and adding in a sprint backlog.
- However, in Kanban, we do not have sprint, so sprint backlog activity will not be performed.
- This is the main difference between scrum and Kanban that scrum contains sprint backlog while kanban does not contain the sprint backlog.

