# MALLA REDDY UNIVERSITY

## R22 - II YEAR / I SEM

## (MR22-1CS0146)

## OBJECT ORIENTED SOFTWARE ENGINEERING

## A. Y. 2023 – 2024

## UNIT–I

**Introduction to Software Engineering:** A generic view of Process: Software Engineering, Process Framework, CMM Process Patterns and Process Assessment.

**Process Models**: Prescriptive Models, Waterfall Model, Incremental Process Models, Evolutionary Process Models, Specialized Process Models, The Unified Models, Personal and Team Process Models, Process Technology, Product and Process.

# INTRODUCTION TO SOFTWARE ENGINEERING

## A GENERIC VIEW OF PROCESS

## SOFTWARE ENGINEERING - A LAYERED TECHNOLOGY:

Software engineering is a layered technology. Any engineering approach must rest on an organizational commitment to quality. **The bedrock that supports software engineering is a quality focus.**

The foundation for software engineering is the process layer. Software engineering process is the glue that holds the technology layers. **Process defines a framework that must be established for effective delivery of software engineering technology**.

The software forms the basis for management control of software projects and establishes the context in which

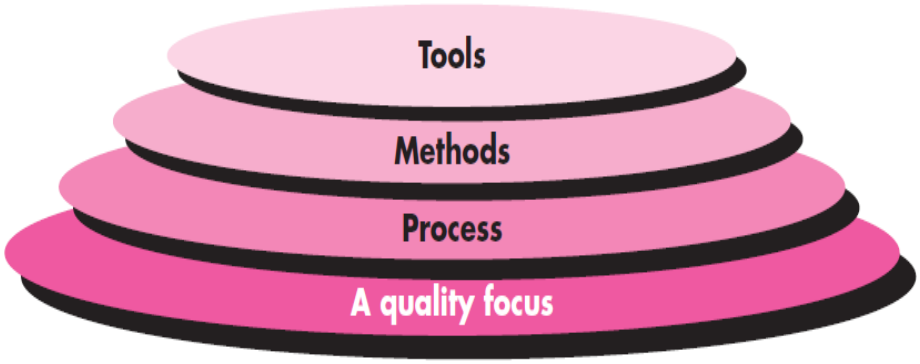
- Technical methods are applied,
- Work products are produced,
- Milestones are established,
- Quality is ensured,
- And change is properly managed.

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

Methods encompass a broad array of tasks that include

- ✓ Communication,
- ✓ Requirements analysis,
- ✓ Design modeling,
- ✓ Program construction,
- ✓ Testing and support.

**Software engineering tools provide automated or semiautomated 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.




**Software Engineering Layers**

## A PROCESS FRAMEWORK:

- **Software process** must be established for effective delivery of software engineering technology.
- A **process framework** establishes the foundation for a complete software process by identifying a small number of framework activities that are applicable to all software projects, regardless of their size or complexity.
- The process framework encompasses a **set of umbrella activities** that are applicable across the entire software process.
- Each **framework activity** is populated by a set of software engineering actions
- Each **software engineering action** is represented by a number of different task sets- each a collection of software engineering work tasks, related work products, quality assurance points, and project milestones.

*In brief*

"A **process** defines who is doing what, when, and how to reach a certain goal."

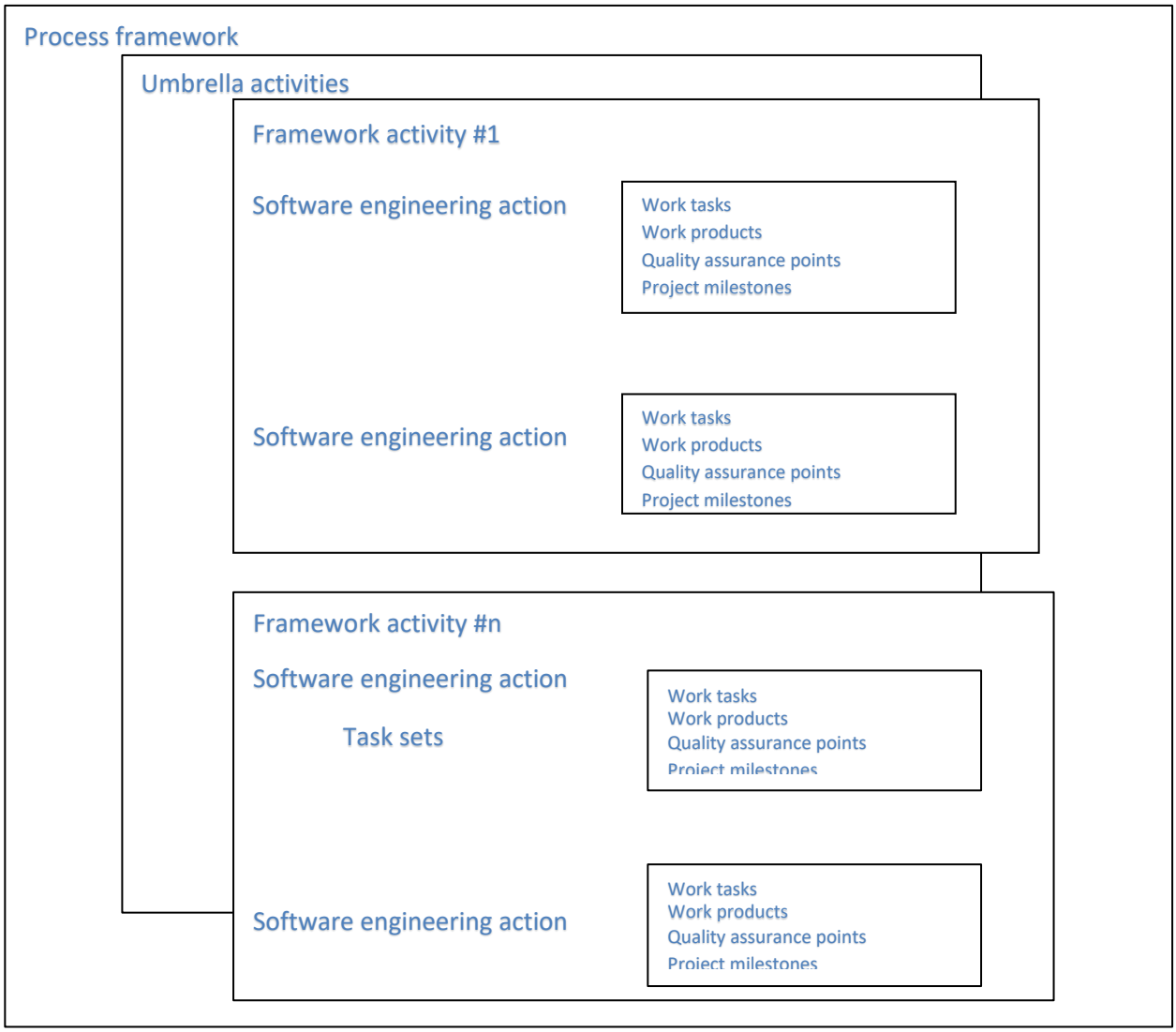
A **Process Framework**

- establishes the foundation for a complete software process
- identifies a small number of **framework activities**
  - applies to all s/w projects, regardless of size/complexity.
- also, set of **umbrella activities**
  - applicable across entire s/w process.

- Each **framework activity** has
  - set of **s/w engineering actions**.

- Each **s/w engineering action** (e.g., design) has
  - collection of related **tasks (**called **task sets):**
    - **work tasks**
    - work products (deliverables)
    - quality assurance points
    - project milestones.

# Software process




**Generic Process Framework**: It is applicable to the vast majority of software projects

- Communication activity
- Planning activity
- Modeling activity
    - analysis action
        - requirements gathering work task
        - elaboration work task
        - negotiation work task
        - specification work task
        - validation work task

- design action
    - data design work task
    - architectural design work task
    - interface design work task
    - component-level design work task
- Construction activity
- Deployment activity

**1) Communication**: This framework activity involves heavy communication and collaboration with the customer and encompasses requirements gathering and other related activities.

**2) Planning**: This activity establishes a plan for the software engineering work that follows. It describes the technical tasks to be conducted, the risks that are likely, the resources that will be required, the work products to be produced, and a work schedule.

**3) Modeling:** This activity encompasses the creation of models that allow the developer and customer to better understand software requirements and the design that will achieve those requirements. The modeling activity is composed of 2 software engineering actions- analysis and design.

- ✓ Analysis encompasses a set of work tasks.
- ✓ Design encompasses work tasks that create a design model.

**4) Construction:** This activity combines core generation and the testing that is required to uncover the errors in the code.

**5) Deployment:** The software is delivered to the customer who evaluates the delivered product and provides feedback based on the evolution.

These 5 generic framework activities can be used during the development of small programs, the creation of large web applications, and for the engineering of large, complex computer-based systems.

The following are the set of **Umbrella Activities.**

**1) Software project tracking and control –** allows the software team to assess progress against the project plan and take necessary action to maintain schedule.

**2) Risk Management -** assesses risks that may effect 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) Formal Technical Reviews** - assesses software engineering work products in an effort to uncover and remove errors before they are propagated to the next action or activity.

**5) Measurement -** define and collects process, project and product measures that assist the team in delivering software that needs customer's 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 management** - defines criteria for work product reuse and establishes mechanisms to achieve reusable components.

8) **Work Product preparation and production** - encompasses the activities required to create work products such as models, document, logs, forms and lists.

Intelligent application of any software process model must recognize that adaption is essential for success but process models do differ fundamentally in:

- ✓ The overall flow of activities and tasks and the interdependencies among activities and tasks.
- ✓ The degree through which work tasks are defined within each frame work activity.
- ✓ The degree through which work products are identified and required.
- ✓ The manner which quality assurance activities are applied.
- ✓ The manner in which project tracking and control activities are applied.
- ✓ The overall degree of the detailed and rigor with which the process is described.
- ✓ The degree through which the customer and other stakeholders are involved with the project.
- ✓ The level of autonomy given to the software project team.
- ✓ The degree to which team organization and roles are prescribed.

## THE CAPABILITY MATURITY MODEL INTEGRATION (CMMI):

The CMMI represents a process meta-model in two different ways:

- As a continuous model
- As a staged model.

**E**ach process area is formally assessed against specific goals and practices and is rated according to the following capability levels.

**Level 0: Incomplete.** The process area is either not performed or does not achieve all goals and objectives defined by CMMI for level 1 capability.

**Level 1: Performed**. All of the specific goals of the process area have been satisfied. Work tasks required to produce defined work products are being conducted.

**Level 2: Managed.** All level 1 criteria have been satisfied. In addition, all work associated with the process area conforms to an organizationally defined policy; all people doing the work have access to adequate resources to get the job done; stakeholders are actively involved in the process area as required; all work tasks and work products are "monitored, controlled, and reviewed;

**Level 3: Defined.** All level 2 criteria have been achieved. In addition, the process is "tailored from the organizations set of standard processes according to the organizations tailoring guidelines, and contributes and work products, measures and other process-improvement information to the organizational process assets".

**Level 4: Quantitatively managed.** All level 3 criteria have been achieved. In addition, the process area is controlled and improved using measurement and quantitative assessment. "Quantitative objectives for quality and process performance are established and used as criteria in managing the process"

**Level 5: Optimized.** All level 4 criteria have been achieved. In addition, the process area is adapted and optimized using quantitative means to meet changing customer needs and to continually improve the efficacy of the process area under consideration"

The CMMI defines each process area in terms of "specific goals" and the "specific practices" required to achieve these goals. Specific practices refine a goal into a set of process-related activities.

**The specific goals (SG)** and the associated specific practices (SP) defined for project planning are

**SG 1 Establish estimates**

SP 1.1 Estimate the scope of the project

SP 1.2 Establish estimates of work product and task attributes SP 1.3 Define project life cycle

SP 1.4 Determine estimates of effort and cost

**SG 2 Develop a Project Plan**

SP 2.1 Establish the budget and schedule SP 2.2 Identify project risks

SP 2.3 Plan for data management

SP 2.4 Plan for needed knowledge and skills SP 2.5 Plan stakeholder involvement

SP 2.6 Establish the project plan

**SG 3 Obtain commitment to the plan**

SP 3.1 Review plans that affect the project SP 3.2 Reconcile work and resource levels SP 3.3 Obtain plan commitment

In addition to specific goals and practices, the CMMI also defines a set of five generic goals and related practices for each process area. Each of the five generic goals corresponds to one of the five capability levels. Hence to achieve a particular capability level, the generic goal for that level and the generic practices that correspond to that goal must be achieved.

To illustrate, **the generic goals (GG) and practices (GP)** for the project planning process area are

**GG 1 Achieve specific goals**

GP 1.1 Perform base practices

**GG 2 Institutionalize a managed process**

GP 2.1 Establish and organizational policy

GP 2.2 Plan the process

GP 2.3 Provide resources

GP 2.4 Assign responsibility

GP 2.5 Train people

GP 2.6 Manage configurations

GP 2.7 Identify and involve relevant stakeholders

GP 2.8 Monitor and control the process

GP 2.9 Objectively evaluate adherence

GP 2.10 Review status with higher level management

**GG 3 Institutionalize a defined process**

GP 3.1 Establish a defined process

GP 3.2 Collect improvement information

**GG 4 Institutionalize a quantitatively managed process**

GP 4.1 Establish quantitative objectives for the process

GP 4.2 Stabilize sub process performance

**GG 5 Institutionalize and optimizing process**

GP 5.1 Ensure continuous process improvement

GP 5.2 Correct root causes of problems

## PROCESS PATTERNS:

The software process can be defined as a collection patterns that define a set of activities, actions, work tasks, work products and/or related behaviors required to develop computer software.

A process pattern provides us with a template- a consistent method for describing an important characteristic of the software process. A pattern might be used to describe a complete process and a task within a framework activity.

**Pattern Name:** The pattern is given a meaningful name that describes its function within the software process.

**Intent:** The objective of the pattern is described briefly.

**Type:** The pattern type is specified. There are three types

1. **Task patterns** define a software engineering action or work task that is part of the process and relevant to successful software engineering practice.
   *Example:* Requirement Gathering
2. **Stage Patterns** define a framework activity for the process. This pattern incorporates multiple task patterns that are relevant to the stage.
   *Example:* Communication
3. **Phase patterns** define the sequence of framework activities that occur with the process, even when the overall flow of activities is iterative in nature.
   *Example:* Spiral model or prototyping.

**Initial Context:** The conditions under which the pattern applies are described prior to the initiation of the pattern, we ask

(1) What organizational or team related activities have already occurred.
(2) What is the entry state for the process
(3) What software engineering information or project information already exists

**Problem:** The problem to be solved by the pattern is described.

**Solution:** The implementation of the pattern is described.

This section describes how the initial state of the process is modified as a consequence the initiation of the pattern.

It also describes how software engineering information or project information that is available before the initiation of the pattern is transformed as a consequence of the successful execution of the pattern

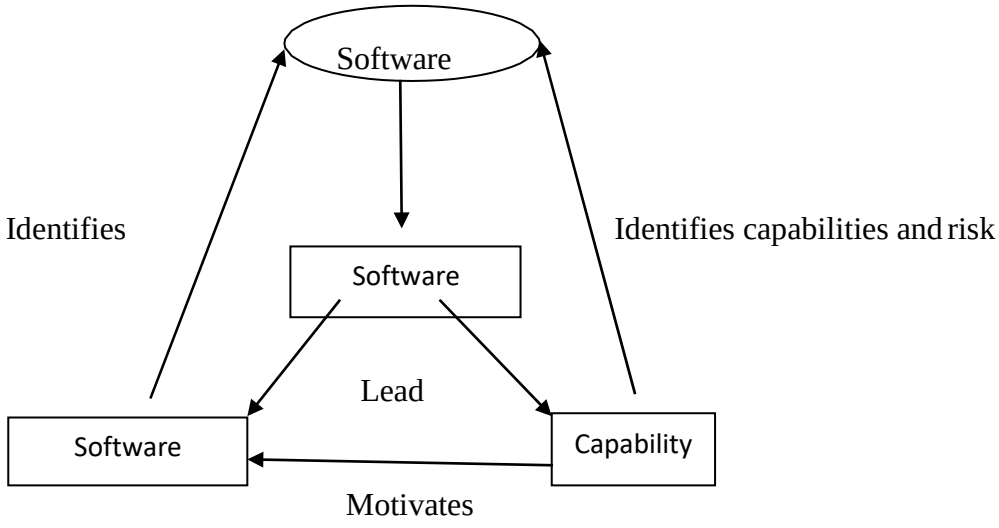
**Resulting Context:** The conditions that will result once the pattern has been successfully implemented are described. Upon completion of the pattern we ask

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

**Known Uses:** The specific instances in which the pattern is applicable are indicated Process patterns provide and effective mechanism for describing any software process.
The patterns enable a software engineering organization to develop a hierarchical process description that begins at a high-level of abstraction.
Once process pattern have been developed, they can be reused for the definition of process variants-that is, a customized process model can be defined by a software team using the pattern as building blocks for the process models.

## PROCESS ASSESSMENT:

The existence of a software process is no guarantee that software will be delivered on time, that it will meet the customer's needs, or that it will exhibit the technical characteristics that will lead to long-term quality characteristics. In addition, the process itself should be assessed to be essential to ensure that it meets a set of basic process criteria that have been shown to be essential for a successful software engineering.




A Number of different approaches to software process assessment have been proposed over the past few decades.

**Standards CMMI Assessment Method for Process Improvement (SCAMPI)** provides a five step process assessment model that incorporates initiating, diagnosing, establishing, acting & learning. The SCAMPI method uses the SEI CMMI as the basis for assessment.

**CMM Based Appraisal for Internal Process Improvement (CBA IPI)** provides a diagnostic technique for assessing the relative maturity of a software organization, using the SEI CMM as the basis for the assessment.

**SPICE (ISO/IEC15504)** standard defines a set of requirements for software process assessments. The intent of the standard is to assist organizations in developing an objective evaluation of the efficacy of any defined software process.

**ISO 9001:2000 for Software** is a generic standard that applies to any organization that wants to improve the overall quality of the products, system, or services that it provides. Therefore, the standard is directly applicable to software organizations &companies.

# PROCESS MODELS

## PRESCRIPTIVE MODELS:

Prescriptive process models were originally proposed to bring order to the chaos of software development. History has indicated that these traditional models have brought a certain amount of useful structure to software engineering work and have provided a reasonably effective road map for software teams. However, software engineering work and the product that it produces remain on "the edge of chaos."

The edge of chaos is defined as "a natural state between order and chaos, a grand compromise between structure and surprise". The edge of chaos can be visualized as an unstable, partially structured state. It is unstable because it is constantly attracted to chaos or to absolute order.

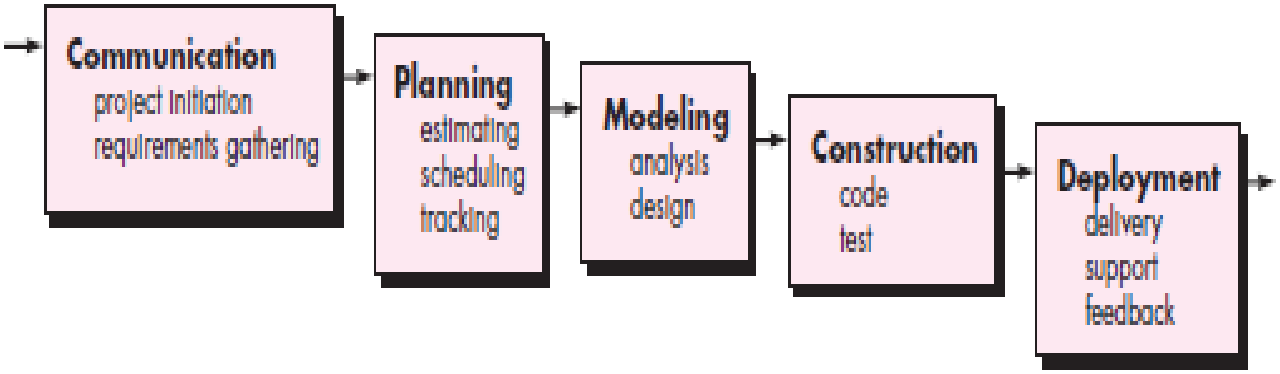
**Prescriptive process models** define a set of activities, actions, tasks, milestones, and work products that are required to engineer high-quality software. These process models are not perfect, but they do provide a useful roadmap for software engineering work.

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

## WATERFALL MODEL:

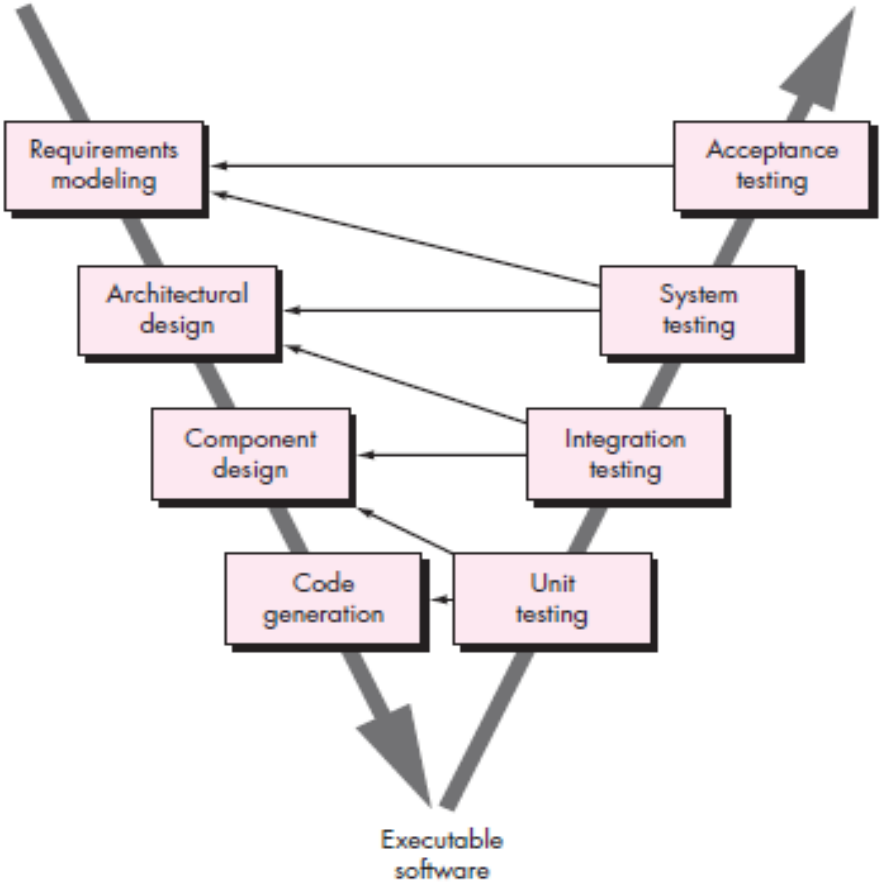
There are times when the requirements for a problem are well understood—when work flows from **communication** through **deployment** in a reasonably linear fashion. This situation is sometimes encountered when well-defined adaptations or enhancements to an existing system must be made (e.g., an adaptation to accounting software that has been mandated because of changes to government regulations). It may also occur in a limited number of new development efforts, but only when requirements are well defined and reasonably stable.

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.



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

The V-model depicts the relationship of quality assurance actions to the actions associated with communication, modeling, and early construction activities. As a software team moves down the left side of the V, basic problem requirements are refined into progressively more detailed and technical representations of the problem and its solution. Once code has been generated, the team moves up the right side of the V, essentially performing a series of tests (quality assurance actions) that validate each of the models created as the team moved down the left side. 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. However, over the past three decades, criticism of this process model has caused even ardent supporters to question its efficacy. 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. 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 exists at the beginning of many projects.

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.

In an interesting analysis of actual projects, Bradac [Bra94] found that the linear nature of the classic life cycle leads to "blocking states" in which some project team members must wait for other members of the team to complete dependent tasks. In fact, the time spent waiting can exceed the time spent on productive work! The blocking states tend to be more prevalent at the beginning and end of a linear sequential process.

Today, software work is fast-paced and subject to a never-ending stream of changes (to features, functions, and information content). The waterfall model is often inappropriate for such work. However, it can serve as a useful process model in situations where requirements are fixed and work is to proceed to completion in a linear manner.

## INCREMENTAL PROCESS MODELS:

There are many situations in which initial software requirements are reasonably well defined, but the overall scope of the development effort precludes a purely linear process. In addition, there may be a compelling need to provide a limited set of software functionality to users quickly and then refine and expand on that functionality in later software releases. In such cases, you can choose a process model that is designed to produce the software in increments.

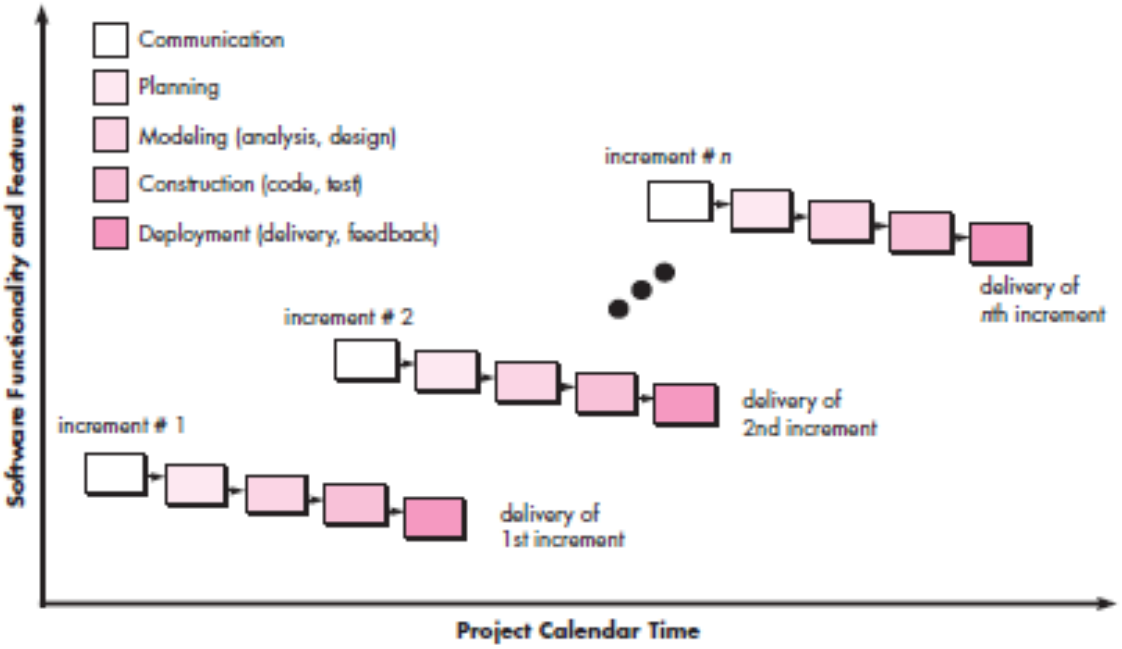
The *incremental* model combines elements of linear and parallel process flows. The incremental model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces deliverable "increments" of the software in a manner that is similar to the increments produced by an evolutionary 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, 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.

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. 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, 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.




## EVOLUTIONARY PROCESS MODELS:

Software, like all complex systems, evolves over a period of time. Business and product requirements often change as development proceeds, making a straight line path to an end product unrealistic; tight market deadlines make completion of a comprehensive software product impossible, but a limited version must be introduced to meet competitive or business pressure; a set of core product or system requirements
is well understood, but the details of product or system extensions have yet to be defined. In these and similar situations, you need a process model that has been explicitly designed to accommodate a product that evolves over time. Evolutionary models are iterative. They are characterized in a manner that enables you to develop increasingly more complete versions of the software.

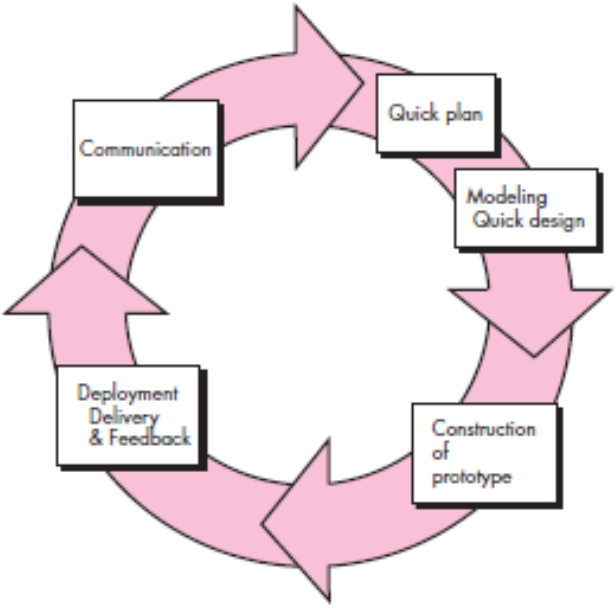
Two common evolutionary process models.

**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.

Although prototyping can be used as a stand-alone process model, it is more commonly used as a technique that can be implemented within the context of any one of the process models. Regardless of the manner in which it is applied, the prototyping paradigm assists you and other stakeholders to better understand what is to be built when requirements are fuzzy. The prototyping

paradigm 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 users (e.g., human interface layout or output display formats). The quick design leads to the construction of a prototype.

The prototype is deployed and evaluated by stakeholders, who provide feedback that is used to further refine requirements. Iteration occurs as the prototype is tuned to satisfy the needs of various stakeholders, while at the same time enabling you to better understand what needs to be done. 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. But what do you do with the prototype when it has served the purpose described earlier? Brooks [Bro95] provides one answer: In most projects, the first system built is barely usable. It may be too slow, too big, awkward in use or all three. There is no alternative but to start again, smarting but smarter, and build a redesigned version in which these problems are solved.




The prototype can serve as "the first system." The one that Brooks recommends you throw away. But this may be an idealized view. Although some prototypes are built as "throwaways," others are evolutionary in the sense that the prototype slowly evolves into the actual system. Both stakeholders and software engineers like the prototyping paradigm. Users get a feel for the actual system, and developers get to build something immediately.

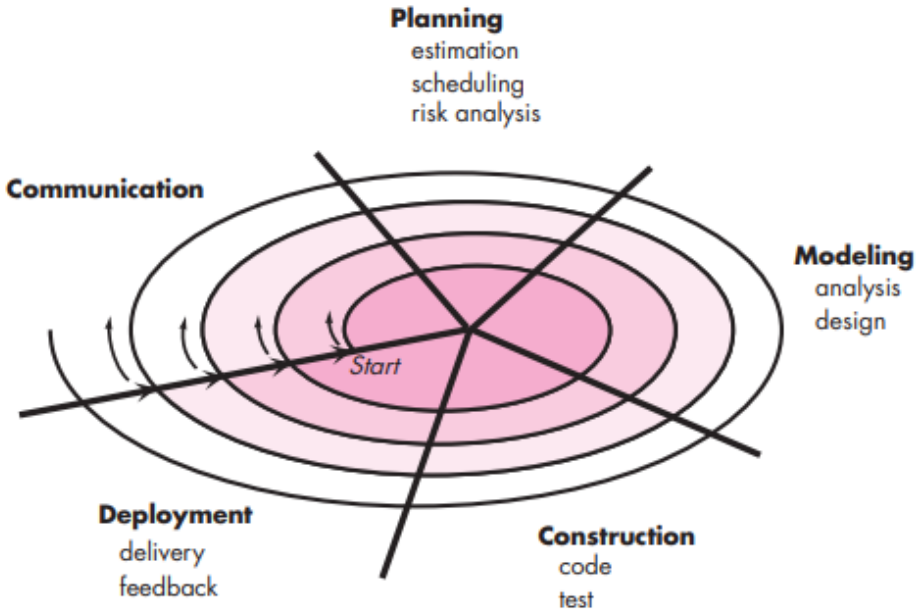
**Yet, prototyping can be problematic for the following reasons:**

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. When informed that the product must be rebuilt so that high levels of quality can be maintained, stakeholders cry foul and demand that "a few fixes" be applied to make the prototype a working product. Too often, software development management relents.

2. As a software engineer, you often make 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, you 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.

Although problems can occur, prototyping can be an effective paradigm for software engineering. The key is to define the rules of the game at the beginning; that is, all stakeholders should agree that the prototype is built to serve as a mechanism for defining requirements. It is then discarded (at least in part), and the actual software is engineered with an eye toward quality.

**The Spiral Model.** Originally proposed by Barry Boehm [Boe88], the spiral model is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model. It provides the potential for rapid development of increasingly more complete versions of the software. Boehm [Boe01a] describes the model in the following manner: The spiral development model is a risk-driven process model generator that is used to guide multi-stakeholder concurrent engineering of software intensive systems. It has two main distinguishing features. One is a cyclic approach for incrementally growing a system's degree of definition and implementation while decreasing its degree of risk. The other is a set of anchor point milestones for ensuring stakeholder commitment to feasible and mutually satisfactory system solutions. Using the spiral model, software is developed in a series of evolutionary releases. During early iterations, the release might be a model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced.




A spiral model is divided into a set of framework activities defined by the software engineering team. Each of the framework activities represent one segment of the spiral path. As this evolutionary process begins, the software team performs activities that are implied by a circuit around the spiral in a clockwise direction, beginning at the center. Risk is considered as each revolution is made. Anchor point milestones—a combination of work products and conditions that are attained along the path of the spiral—are noted for each evolutionary pass. The first circuit around the spiral might result in the

development of a product specification; subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software. Each pass through the planning region results in adjustments to the project plan. Cost and schedule are adjusted based on feedback derived from the customer after delivery. In addition, the project manager adjusts the planned number of iterations required to complete the software. Unlike other process models that end when software is delivered, the spiral model can be adapted to apply throughout the life of the computer software. Therefore, the first circuit around the spiral might represent a "concept development project" that starts at the core of the spiral and continues for multiple iterations until concept development is complete.

If the concept is to be developed into an actual product, the process proceeds outward on the spiral and a "new product development project" commences. The new product will evolve through a number of iterations around the spiral. Later, a circuit around the spiral might be used to represent a "product enhancement project." In essence, the spiral, when characterized in this way, remains operative until the software is retired. There are times when the process is dormant, but whenever a change is initiated, the process starts at the appropriate entry point (e.g., product enhancement).

The spiral model is a realistic approach to the development of large-scale systems and software. Because software evolves as the process progresses, the developer and customer better understand and react to risks at each evolutionary level. The spiral model uses prototyping as a risk reduction mechanism but, more important, enables you to apply the prototyping approach at any stage in the evolution of the product. It maintains the systematic stepwise approach suggested by the classic life cycle but incorporates it into an iterative framework that more realistically reflects the real world.

The spiral model demands a direct consideration of technical risks at all stages of the project and, if properly applied, should reduce risks before they become problematic. But like other paradigms, the spiral model is not a panacea. It may be difficult to convince customers (particularly in contract situations) 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.

## SPECIALIZED PROCESS MODELS:

Specialized process models take on many of the characteristics of one or more of the traditional models presented in the preceding sections. However, these models tend to be applied when a specialized or narrowly defined software engineering approach is chosen.

### 1. Component-Based Development

Commercial off-the-shelf (COTS) software components, developed by vendors who offer them as products, provide targeted functionality with well-defined interfaces that enable the component to be integrated into the software that is to be built. The component-based development model incorporates many of the characteristics of the spiral model. It is evolutionary in nature, demanding an iterative approach to the creation of software. However, the component-based development model constructs applications from prepackaged software components.

Modeling and construction activities begin with the identification of candidate components. These components can be designed as either conventional software modules or object-oriented classes or packages of classes.

Regardless of the technology that is used to create the components, the component-based development model incorporates the following steps (implemented using an evolutionary approach):

1. Available component-based products are researched and evaluated for the application domain in question.

2. Component integration issues are considered.

3. A software architecture is designed to accommodate the components.

4. Components are integrated into the architecture.

5. Comprehensive testing is conducted to ensure proper functionality.

The component-based development model leads to software reuse, and reusability provides software engineers with a number of measurable benefits. Your software engineering team can achieve a reduction in development cycle time as well as a reduction in project cost if component reuse becomes part of your culture.

## 2. The Formal Methods Model

The formal methods model encompasses a set of activities that leads to formal mathematical specification of computer software. Formal methods enable you to specify, develop, and verify a computer-based system by applying a rigorous, mathematical notation. A variation on this approach, called cleanroom software engineering [Mil87, Dye92], is currently applied by some software development organizations.

When formal methods are used during development, they provide a mechanism for eliminating many of the problems that are difficult to overcome using other software engineering paradigms. Ambiguity, incompleteness, and inconsistency can be discovered and corrected more easily—not through ad hoc review, but through the application of mathematical analysis.

When formal methods are used during design, they serve as a basis for program verification and therefore enable you to discover and correct errors that might otherwise go undetected.

Although not a mainstream approach, the formal methods model offers the promise of defect-free software. Yet, concern about its applicability in a business environment has been voiced:

• The development of formal models is currently quite time consuming and expensive.

• Because few software developers have the necessary background to apply formal methods, extensive training is required.

• It is difficult to use the models as a communication mechanism for technically unsophisticated customers.

These concerns notwithstanding, the formal methods approach has gained adherents among software developers who must build safety-critical software (e.g., developers of aircraft avionics and medical devices) and among developers that would suffer severe economic hardship should software errors occur.

## 3. Aspect-Oriented Software Development

Regardless of the software process that is chosen, the builders of complex software invariably implement a set of localized features, functions, and information content. These localized software characteristics are modeled as components (e.g., objectoriented classes) and then constructed within the context of a system architecture. As modern computer-based systems become more sophisticated (and complex), certain concerns—customer required properties or areas of technical interest—span the entire architecture. Some concerns are high-level properties of a system (e.g., security, fault tolerance). Other concerns affect functions (e.g., the application of business rules), while others are systemic (e.g., task synchronization or memory management).

When concerns cut across multiple system functions, features, and information, they are often referred to as crosscutting concerns. Aspectual requirements define those crosscutting concerns that have an impact across the software architecture. Aspect-oriented software development (AOSD), often referred to as aspect-oriented programming (AOP), is a relatively new software engineering paradigm that provides a process and methodological approach for defining, specifying, designing, and constructing aspects - "mechanisms beyond subroutines and inheritance for localizing the expression of a crosscutting concern".

Grundy provides further discussion of aspects in the context of what he calls aspect-oriented component engineering (AOCE):

AOCE uses a concept of horizontal slices through vertically-decomposed software components, called "aspects," to characterize cross-cutting functional and non-functional properties of components. Common, systemic aspects include user interfaces, collaborative work, distribution, persistency, memory management, transaction processing, security, integrity and so on. Components may provide or require one or more "aspect details" relating to a particular aspect, such as a viewing mechanism, extensible affordance and interface kind (user interface aspects); event generation, transport and receiving (distribution aspects); data store/retrieve and indexing (persistency aspects); authentication, encoding and access rights (security aspects); transaction atomicity, concurrency control and logging strategy (transaction aspects); and so on. Each aspect detail has a number of properties, relating to functional and/or non-functional characteristics of the aspect detail.

A distinct aspect-oriented process has not yet matured. However, it is likely that such a process will adopt characteristics of both evolutionary and concurrent process models. The evolutionary model is appropriate as aspects are identified and then constructed. The parallel nature of concurrent development is essential because aspects are engineered independently of localized software components and yet, aspects have a direct impact on these components. Hence, it is essential to instantiate asynchronous communication between the software process activities applied to the engineering and construction of aspects and components.

# THE UNIFIED PROCESS:

In the seminal book on the Unified Process, Ivar Jacobson, Grady Booch, and James Rumbaugh discuss the need for a "use case driven, architecture-centric, iterative and incremental" software process when they state:

Today, the trend in software is toward bigger, more complex systems. That is due in part to the fact that computers become more powerful every year, leading users to expect more from them. This trend has also been influenced by the expanding use of the Internet for exchanging all kinds of information. Our appetite for ever-more sophisticated software grows as we learn from one product release to the next how the product could be improved. We want software that is better adapted to our needs, but that, in turn, merely makes the software more complex. In short, we want more.

In some ways the Unified Process is an attempt to draw on the best features and characteristics of traditional software process models, but characterize them in a way that implements many of the best principles of agile software development.

The Unified Process recognizes the importance of customer communication and streamlined methods for describing the customer's view of a system. It emphasizes the important role of software architecture and "helps the architect focus on the right goals, such as understandability, reliance to future changes, and reuse".

It suggests a process flow that is iterative and incremental, providing the evolutionary feel that is essential in modern software development.

## 1. A Brief History

During the early 1990s James Rumbaugh [Rum91], Grady Booch [Boo94], and Ivar Jacobson [Jac92] began working on a "unified method" that would combine the best features of each of their individual object-oriented analysis and design methods and adopt additional features proposed by other experts (e.g., [Wir90]) in object-oriented modeling.

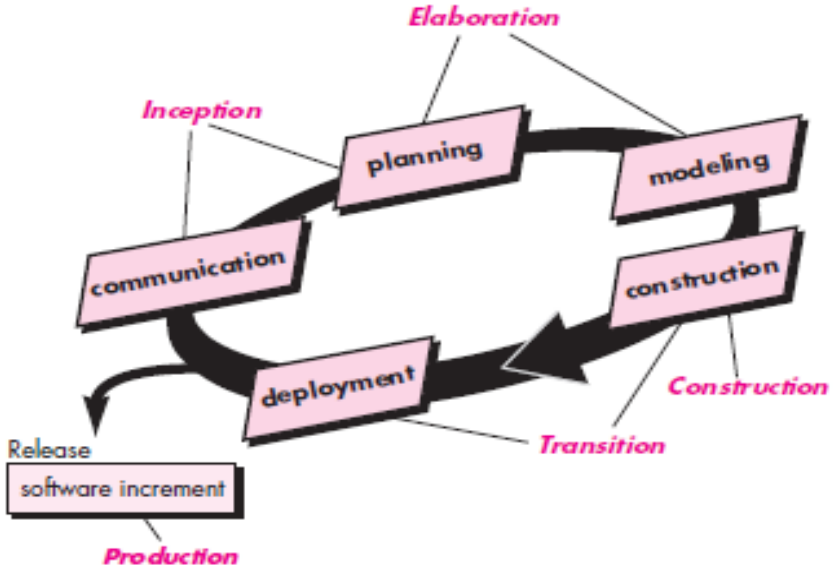
The result was UML—a unified modeling language that contains a robust notation for the modeling and development of object-oriented systems. By 1997, UML became a de facto industry standard for object-oriented software development.

UML is used to represent both requirements and design models. UML provided the necessary technology to support object-oriented software engineering practice, but it did not provide the process framework to guide project teams in their application of the technology. Over the next few years, Jacobson, Rumbaugh, and Booch developed the Unified Process, a framework for object-oriented software engineering using UML.

Today, the Unified Process (UP) and UML are widely used on object-oriented projects of all kinds. The iterative, incremental model proposed by the UP can and should be adapted to meet specific project needs.

## 2. Phases of the Unified Process

The below figure depicts the "phases" of the UP and relates them to the generic activities.




**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.

Fundamental business requirements are described through a set of preliminary use cases that describe which features and functions each major class of users desires. 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. Planning identifies resources, assesses major risks, defines a schedule, and establishes a basis for the phases that are to be applied as the software increment is developed.

**The elaboration phase** encompasses the communication and modeling activities of the generic process model. Elaboration refines and expands the preliminary use cases that were developed as part of the inception phase and expands the architectural representation to include five different views of the software - the use case model, the requirements model, the design model, the implementation model, and the deployment model.

In some cases, elaboration creates an "executable architectural baseline" that represents a "first cut" executable system.

The architectural baseline demonstrates the viability of the architecture but does not provide all features and functions required to use the system. In addition, the plan is carefully reviewed at the culmination of the elaboration phase to ensure that scope, risks, and delivery dates remain reasonable. Modifications to the plan are often made at this time.

**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. All necessary and required features and functions for the software increment (i.e., the release) are then implemented in source code. As components are being implemented, unit tests are designed and executed for each. In addition, integration activities (component assembly and integration testing) are conducted. Use cases are used to derive a suite of acceptance tests that are executed prior to the initiation of the next UP phase.

**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. At the conclusion of the transition phase, the software increment becomes a usable software 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. It is likely that at the same time the construction, transition, and production phases are being conducted, work may have already begun on the next software increment. This means that the five UP phases do not occur in a sequence, but rather with staggered concurrency.

A software engineering workflow is distributed across all UP phases. In the context of UP, a workflow is analogous to a task set (described earlier in this chapter). That is, a workflow identifies the tasks required to accomplish an important software engineering action and the work products that are produced as a consequence of successfully completing the tasks. It should be noted that not every task identified for a UP workflow is conducted for every software project. The team adapts the process (actions, tasks, subtasks, and work products) to meet its needs.

## PERSONAL AND TEAM PROCESS MODELS:

The best software process is one that is close to the people who will be doing the work. If a software process model has been developed at a corporate or organizational level, it can be effective only if it is amenable to significant adaptation to meet the needs of the project team that is actually doing software engineering work.

In an ideal setting, you would create a process that best fits your needs, and at the same time, meets the broader needs of the team and the organization. Alternatively, the team itself can create its own process, and at the same time meet the narrower needs of individuals and the broader needs of the organization. Watts Humphrey ([Hum97] and [Hum00]) argues that it is possible to create a “personal software process” and/or a “team software process.” Both require hard work, training, and coordination, but both are achievable.

### 1. Personal Software Process (PSP)

Every developer uses some process to build computer software. The process may be haphazard or ad hoc; may change on a daily basis; may not be efficient, effective, or even successful; but a “process” does exist. Watts Humphrey [Hum97] suggests that in order to change an ineffective personal process, an individual must move through four phases, each requiring training and careful instrumentation.

The Personal Software Process (PSP) emphasizes personal measurement of both the work product that is produced and the resultant quality of the work product. In addition PSP makes the practitioner responsible for project planning (e.g., estimating and scheduling) and empowers the practitioner to control the quality of all software work products that are developed.

The PSP model defines five framework activities:

**Planning.** This activity isolates requirements and develops both size and resource estimates. In addition, a defect estimate (the number of defects projected for the work) is made. All metrics are recorded on worksheets or templates. Finally, development tasks are identified and a project schedule is created.

**High-level design.** External specifications for each component to be constructed are developed and a component design is created. Prototypes are built when uncertainty exists. All issues are recorded and tracked.

**High-level design review.** Formal verification methods are applied to uncover errors in the design. Metrics are maintained for all important tasks and work results.

**Development.** The component-level design is refined and reviewed. Code is generated, reviewed, compiled, and tested. Metrics are maintained for all important tasks and work results.

**Postmortem.** Using the measures and metrics collected (this is a substantial amount of data that should be analyzed statistically), the effectiveness of the process is determined. Measures and metrics should provide guidance for modifying the process to improve its effectiveness.

PSP stresses the need to identify errors early and, just as important, to understand the types of errors that you are likely to make. This is accomplished through a rigorous assessment activity performed on all work products you produce.

PSP represents a disciplined, metrics-based approach to software engineering that may lead to culture shock for many practitioners. However, when PSP is properly introduced to software engineers, the resulting improvement in software engineering productivity and software quality are significant. However, PSP has not been widely adopted throughout the industry. The reasons, sadly, have more to do with human nature and organizational inertia than they do with the strengths and weaknesses of the PSP approach. PSP is intellectually challenging and demands a level of commitment (by practitioners and their managers) that is not always possible to obtain.

Training is relatively lengthy, and training costs are high. The required level of measurement is culturally difficult for many software people. Can PSP be used as an effective software process at a personal level? The answer is an unequivocal “yes.” But even if PSP is not adopted in its entirely, many of the personal process improvement concepts that it introduces are well worth learning.

## 2. Team Software Process (TSP)

Because many industry-grade software projects are addressed by a team of practitioners, Watts Humphrey extended the lessons learned from the introduction of PSP and proposed a Team Software Process (TSP). The goal of TSP is to build a “self-directed” project team that organizes itself to produce high-quality software.

Humphrey defines the following objectives for TSP:

• Build self-directed teams that plan and track their work, establish goals, and own their processes and plans. These can be pure software teams or integrated product teams (IPTs) of 3 to about 20 engineers.

• Show managers how to coach and motivate their teams and how to help them sustain peak performance.

• Accelerate software process improvement by making CMM Level 5 behavior normal and expected.

• Provide improvement guidance to high-maturity organizations.

• Facilitate university teaching of industrial-grade team skills.

A self-directed team has a consistent understanding of its overall goals and objectives; defines roles and responsibilities for each team member; tracks quantitative project data (about productivity and quality); identifies a team process that is appropriate for the project and a strategy for implementing the process; defines local standards that are applicable to the team's software engineering work; continually assesses risk and reacts to it; and tracks, manages, and reports project status.

**TSP defines the following framework activities:**

Project launch, high-level design, implementation, integration and test, and postmortem.

Like their counterparts in PSP (note that terminology is somewhat different), these activities enable the team to plan, design, and construct software in a disciplined manner while at the same time quantitatively measuring the process and the product. The postmortem sets the stage for process improvements.

TSP makes use of a wide variety of scripts, forms, and standards that serve to guide team members in their work. "Scripts" define specific process activities (i.e., project launch, design, implementation, integration and system testing, postmortem) and other more detailed work functions (e.g., development planning, requirements development, software configuration management, unit test) that are part of the team process. TSP recognizes that the best software teams are self-directed.

Team members set project objectives, adapt the process to meet their needs, control the project schedule, and through measurement and analysis of the metrics collected, work continually to improve the team's approach to software engineering.

Like PSP, TSP is a rigorous approach to software engineering that provides distinct and quantifiable benefits in productivity and quality. The team must make a full commitment to the process and must undergo thorough training to ensure that the approach is properly applied.

## PROCESS TECHNOLOGY:

One or more of the process models discussed in the preceding sections must be adapted for use by a software team. To accomplish this, process technology tools have been developed to help software organizations analyze their current process, organize work tasks, control and monitor progress, and manage technical quality. Process technology tools allow a software organization to build an automated model of the process framework, task sets, and umbrella activities.

The model, normally represented as a network, can then be analyzed to determine typical workflow and examine alternative process structures that might lead to reduced development time or cost. Once an acceptable process has been created, other process technology tools can be used to allocate, monitor, and even control all software engineering activities, actions, and tasks defined as part of the process model. Each member of a software team can use such tools to develop a checklist of work tasks

to be performed, work products to be produced, and quality assurance activities to be conducted. The process technology tool can also be used to coordinate the use of other software engineering tools that are appropriate for a particular work task.

## PRODUCT AND PROCESS:

If the process is weak, the end product will undoubtedly suffer. But an obsessive overreliance on process is also dangerous. In a brief essay written many years ago, Margaret Davis [Dav95a] makes timeless comments on the duality of product and process:

About every ten years give or take five, the software community redefines "the problem" by shifting its focus from product issues to process issues. Thus, we have embraced structured programming languages (product) followed by structured analysis methods (process) followed by data encapsulation (product) followed by the current emphasis on the Software Engineering Institute's Software Development Capability Maturity Model (process) [followed by object-oriented methods, followed by agile software development].

While the natural tendency of a pendulum is to come to rest at a point midway between two extremes, the software community's focus constantly shifts because new force is applied when the last swing fails. These swings are harmful in and of themselves because they confuse the average software practitioner by radically changing what it means to perform the job let alone perform it well. The swings also do not solve "the problem" for they are doomed to fail as long as product and process are treated as forming a dichotomy instead of a duality.

There is precedence in the scientific community to advance notions of duality when contradictions in observations cannot be fully explained by one competing theory or another. The dual nature of light, which seems to be simultaneously particle and wave, has been accepted since the 1920s when Louis de Broglie proposed it. I believe that the observations we can make on the artifacts of software and its development demonstrate a fundamental duality between product and process. You can never derive or understand the full artifact, its context, use, meaning, and worth if you view it as only a process or only a product.

All of human activity may be a process, but each of us derives a sense of self-worth from those activities that result in a representation or instance that can be used or appreciated either by more than one person, used over and over, or used in some other context not considered. That is, we derive feelings of satisfaction from reuse of our products by ourselves or others.

Thus, while the rapid assimilation of reuse goals into software development potentially increases the satisfaction software practitioners derive from their work, it also increases the urgency for acceptance of the duality of product and process. Thinking of a reusable artifact as only product or only process either obscures the context and ways to use it or obscures the fact that each use results in product that will, in turn, be used as input to some other software development activity. Taking one view over the other dramatically reduces the opportunities for reuse and, hence, loses the opportunity for increasing job satisfaction.

People derive as much (or more) satisfaction from the creative process as they do from the end product. An artist enjoys the brush strokes as much as the framed result. A writer enjoys the search for the proper metaphor as much as the finished book. As creative software professional, you should also derive as much satisfaction from the process as the end product. The duality of product and process is one important element in keeping creative people engaged as software engineering continues to evolve.