MSC(COMPUTER SCIENCE)
I YEAR I SEMESTER
COURSE TITLE: SOFTWARE ENGINEERING

BY

T. SRILATHA

MSC(IS), MTECH(CSE), SET,(PHD)

ASSISTANT PROFESSOR

RBVRR WOMEN'S COLLEGE

(AUTONOMOUS)

NARAYANAGUDA, HYDERABAD

SOFTWARE ENGINEERING

Contents

Toomula Srilatha,
MSc, MTech SET (PhD)
Assistant Professor, RBVRR Women's College, Hyd

Objectives

- Define the term Software Engineering
- Understand the software process
- Differentiate the software process models
- Identify the requirements for applying Prescriptive process models and Specialized process models
- Software Team Structures in the success of the project
- Role of Software Engineering in the Cloud

SOFTWARE ENGINEERING

Chapter1: Nature of the Software

- 1.1 The Nature of Software
  - 1.1.1 Defining Software
  - 1.1.2 Software Application Domains
  - 1.1.3 Legacy Software
- 1.2 The Changing Nature of Software
  - 1.2.1 Web Apps
  - 1.2.2 Mobile Applications
  - 1.2.3 Cloud Computing
  - 1.2.4 Product Line Software

Toomula Srilatha,
MSc, MTech SET (PhD)
Assistant Professor, RBVRR Women's College, Hyd

Defining Software

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

Definition:

Software is defined as:

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

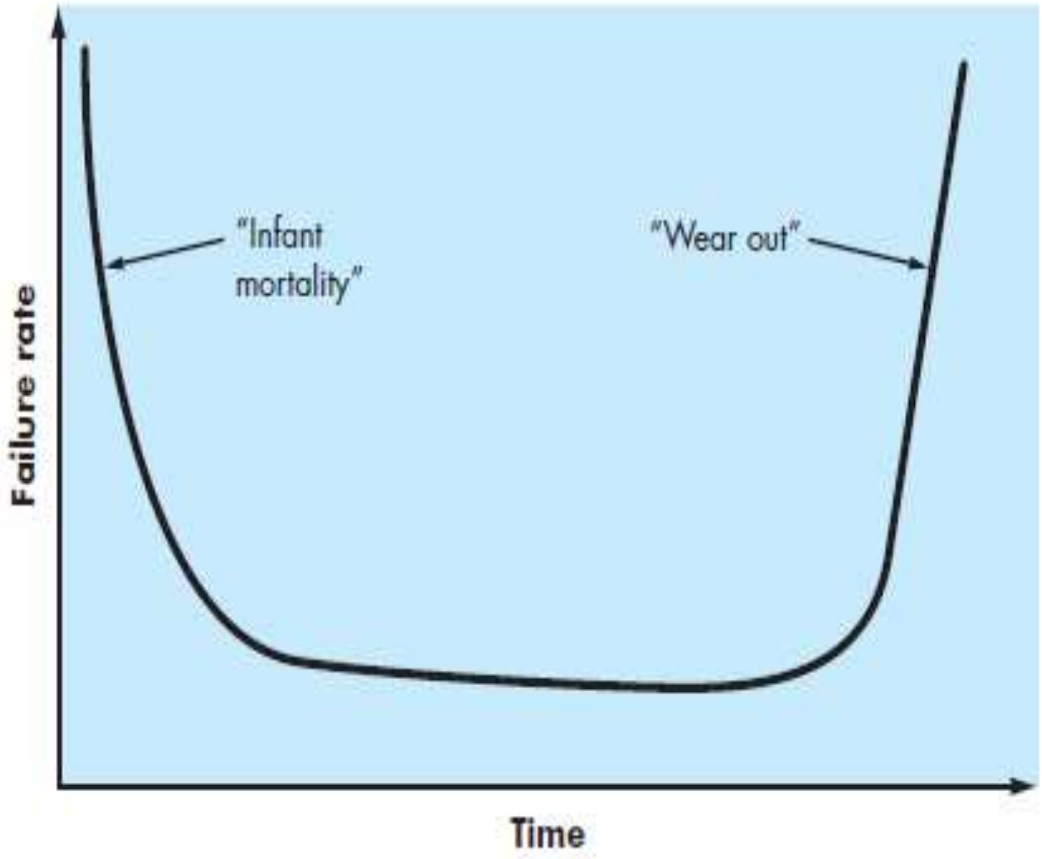
(2) **data structures** that enable the programs to adequately manipulate information, and

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

Defining Software

FIGURE 1.1

Failure curve for hardware



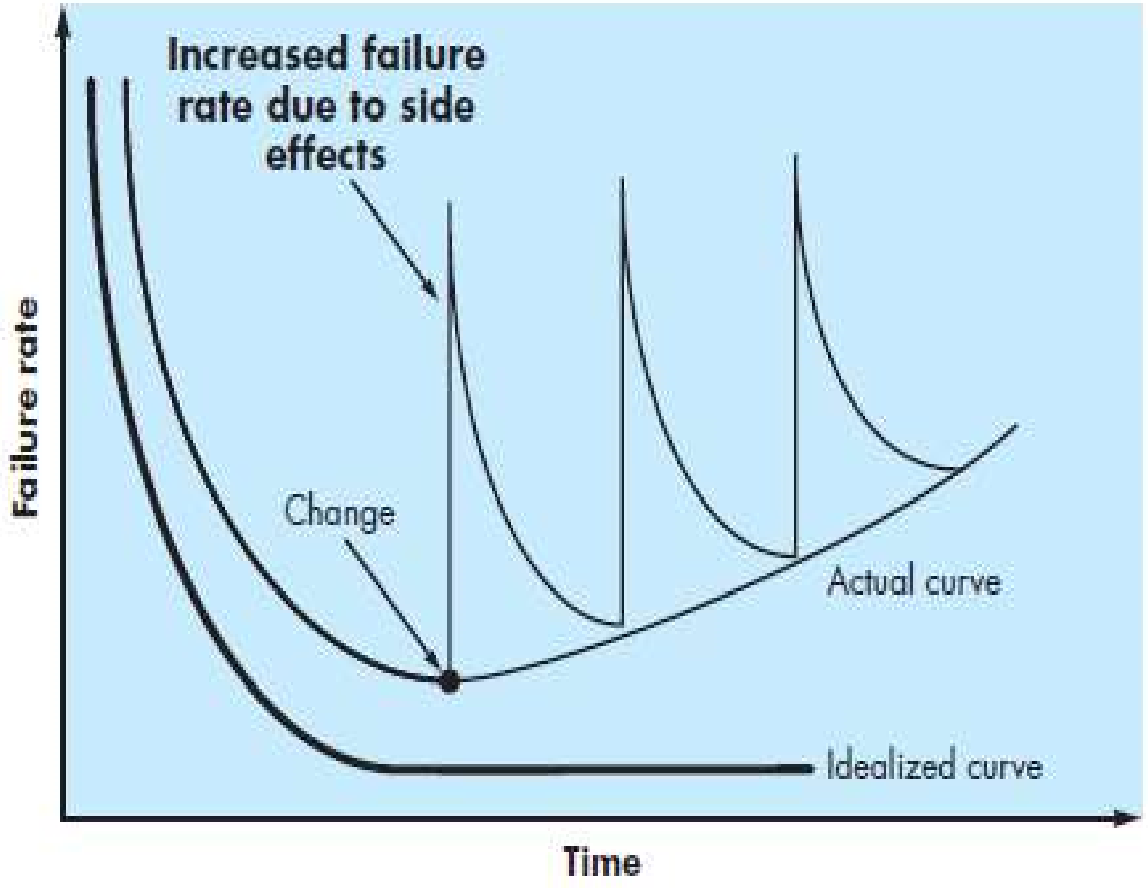
ADVICE

If you want to reduce software deterioration, you'll have to do better software design (Chapters 12 to 18).

Defining Software

FIGURE 1.2

Failure curves for software



Software Application Domains

There are Seven broad categories of computer software is present for challenging software engineers:

1. **System software— a collection of programs written to service other** programs. e.g., compilers, editors, and file management utilities.

- Other systems applications - e.g., operating system components, drivers, networking software, telecommunications processors

2. **Application software —stand-alone programs that solve a specific business need.**

- Applications in this area process business or technical data

Software Application Domains

3. Engineering/scientific software —a broad array of "number-crunching

- programs that range from astronomy to volcanology, from automotive stress analysis to orbital dynamics, and from computer-aided design to molecular biology, from genetic analysis to meteorology.

4. Embedded software— resides within a product or system and is used to

- implement and control features and functions for the end user and for the system itself. (e.g., key pad control for a microwave oven)
- or provide significant function and control capability (e.g., digital functions in an automobile such as fuel control, dashboard displays, and braking systems).

Software Application Domains

5. Product-line software —designed to provide a specific capability for use by many different customers.

- e.g., inventory control products

6. Web/Mobile applications —this network-centric software category spans a wide array of applications and encompasses both browser-based apps and software that resides on mobile devices.

7. Artificial intelligence software— makes use of non-numerical algorithms to

- solve complex problems that are not amenable to computation or straightforward analysis.
- Applications within this area include robotics, expert systems, pattern recognition (image and voice), artificial neural networks, theorem proving, and game playing.

1.1.3 Legacy Software

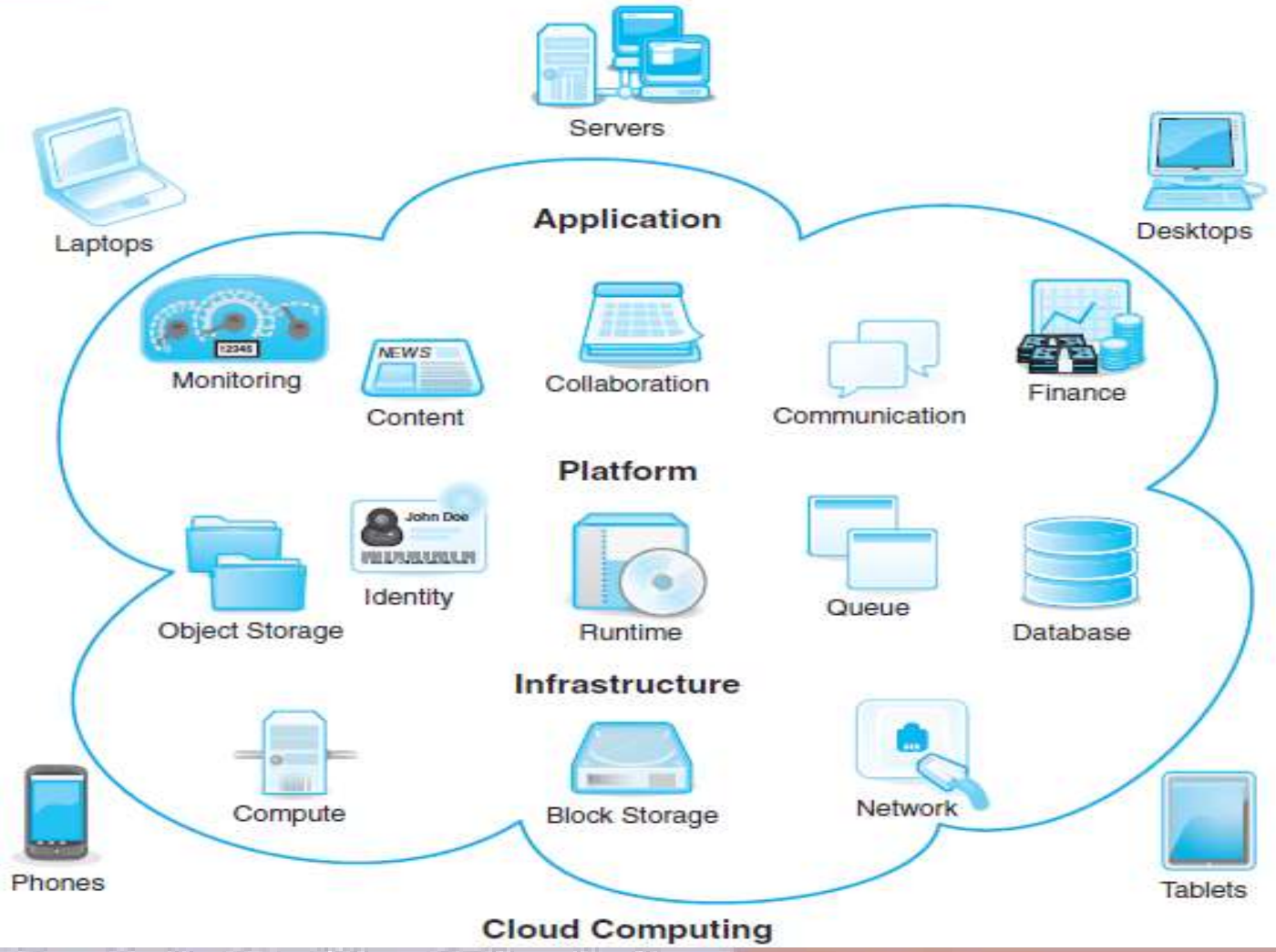
- The older programs, often referred to as *legacy software (*since 1960s).
- Dayani-Fard and his colleagues describe legacy software in the following way: Legacy software systems *. . . were developed decades ago and have been continually* modified to meet changes in business requirements and computing platforms.
- Legacy systems often evolve for one or more of the following reasons:
  - The software must be adapted to meet the needs of new computing environments or technology.
  - The software must be enhanced to implement new business requirements.
  - The software must be extended to make it interoperable with modern systems or databases.
  - The software must be re-architected to make it viable within a evolving computing environment

1.2 THE CHANGING NATURE OF SOFTWARE

- Four broad categories of software are evolving to dominate the industry.
- **WebApps:** The W3C Web Applications (**WebApps**) Working Group is chartered to create specifications that enable improved client-side application development on the Web, including specifications both for application programming interfaces (APIs) for client-side development and for markup vocabularies
- **Mobile Applications:** A **mobile application** (also called a **mobile app**) is a type of **application** designed to run on a **mobile** device, which can be a smartphone or tablet computer
- **Cloud Computing: Cloud computing** is the on-demand availability of **computer** system resources, especially data storage (**cloud** storage) and **computing** power, without direct active management by the user.

Product line Software: The Software Engineering Institute(SEI) defines a *software product line as " a set of* software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment that are developed from a common set of core assets in a prescribed way."

FIGURE 1.3 Cloud computing logical architecture [Wik13]



Chapter – 2
Software Engineering

- 2.1 Defining the Discipline
- 2.2 The Software Process
  - 2.2.1 The Process Framework
  - 2.2.2 Umbrella Activities
  - 2.2.3 Process Adaptation
- 2.3 Software Engineering Practice
  - 2.3.1 The Essence of Practice
  - 2.3.2 General Principles

2.1 DEFINING THE DISCIPLINE

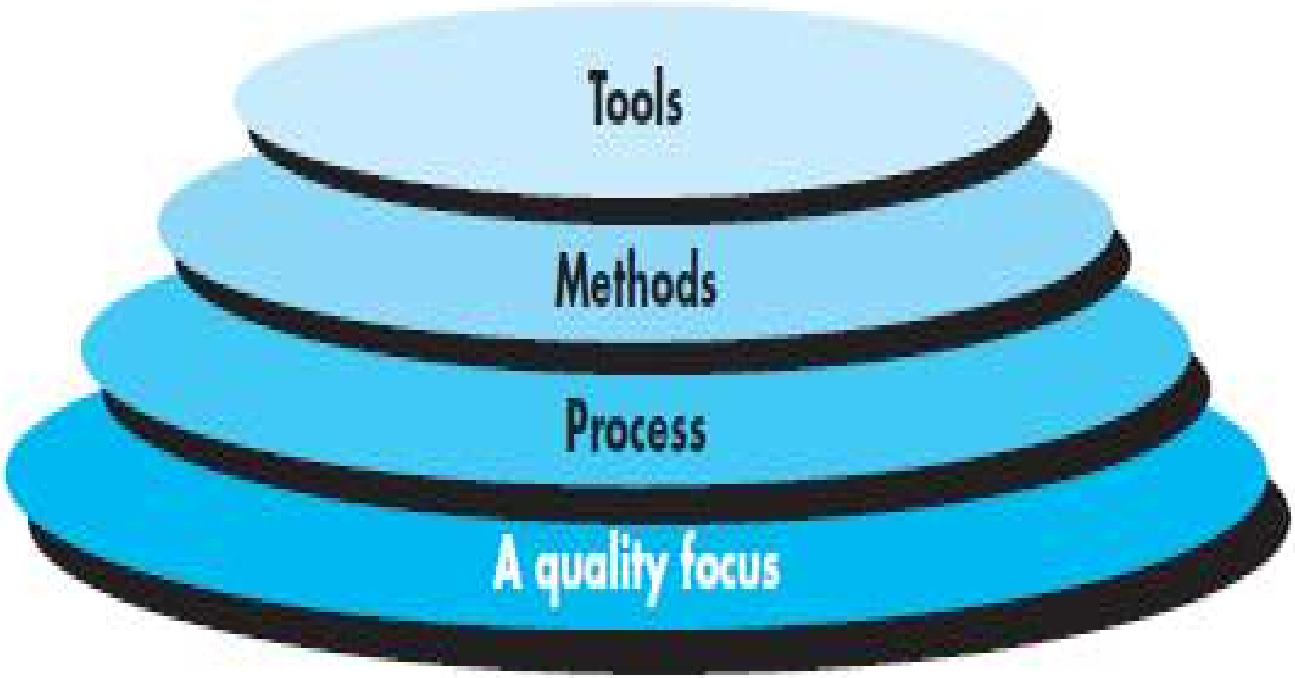
- The IEEE definition for software engineering:

(1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software

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

FIGURE 2.1

Software engineering layers



2.2 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).
- An ***action*** *(e.g., architectural* design) encompasses a set of tasks that produce a major work product (e.g., an architectural model).
- A ***task*** *focuses on a small, but well-defined objective (e.g.,* conducting a unit test) that produces a tangible outcome.

2.2.1 The Process Framework

- A ***process framework*** *identifies* 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.
- A **generic process framework** encompasses **five activities**:
  - **Communication**. Before any technical work can commence, it is critically important
  - **Planning.** Any complicated journey can be simplified if a map exists.
  - **Modeling.** Whether you're a landscaper, a bridge builder, an aeronautical engineer, modeling is very important
  - **Construction.** What you design must be built. This activity combines code generation
  - **Deployment.** The software (as a complete entity or as a partially completed increment)

2.2.2 Umbrella Activities

- Software engineering process framework activities are complemented by a number of *umbrella activities, which are applied throughout* a software project and help a software team manage and control progress, quality, change, and risk.
- Typical umbrella activities include:
  - Software project tracking and control
  - Risk management
  - Software quality assurance.
  - Technical reviews
  - Measurement
  - Software configuration management
  - Reusability management
  - Work product preparation and production

2.2.3 Process Adaptation

- A process adopted for one project might be different than a process adopted for another project.
- The differences are:
  - Overall flow of activities, actions, tasks and the interdependencies
  - Degree to which actions & tasks are defined in each framework activity.
  - Degree to which work products are identified & required.
  - Manner in which quality assurance activities are applied.
  - Manner in which project tracking and control activities are applied.
  - Overall degree of detail with which the process is described.
  - Degree to which the customer and other stakeholders are involved
  - Level of autonomy given to the software team.
  - Degree to which team organization and roles are prescribed.

2.3 SOFTWARE ENGINEERING PRACTICE

- **2.3.1 The Essence of Practice**

George Polya outlined the essence of problem solving in software engineering 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).*

2.3.2 General Principles

- **The First Principle:** *The Reason It All Exists*
- **The Second Principle:** *KISS (Keep It Simple, Stupid!)*
- **The Third Principle:** *Maintain the Vision*
- **The Fourth Principle:** *What You Produce, Others Will Consume*
- **The Fifth Principle:** *Be Open to the Future*
- **The Sixth Principle:** *Plan Ahead for Reuse*
- **The Seventh Principle:** *Think!*

2.4 SOFTWARE DEVELOPMENT MYTHS

- Study and Understand SOFTWARE DEVELOPMENT MYTHS :
  - ❑ Management Myths
  - ❑ Customer Myths
  - ❑ Practitioners myths

Chapter – 3
Software Process Structure

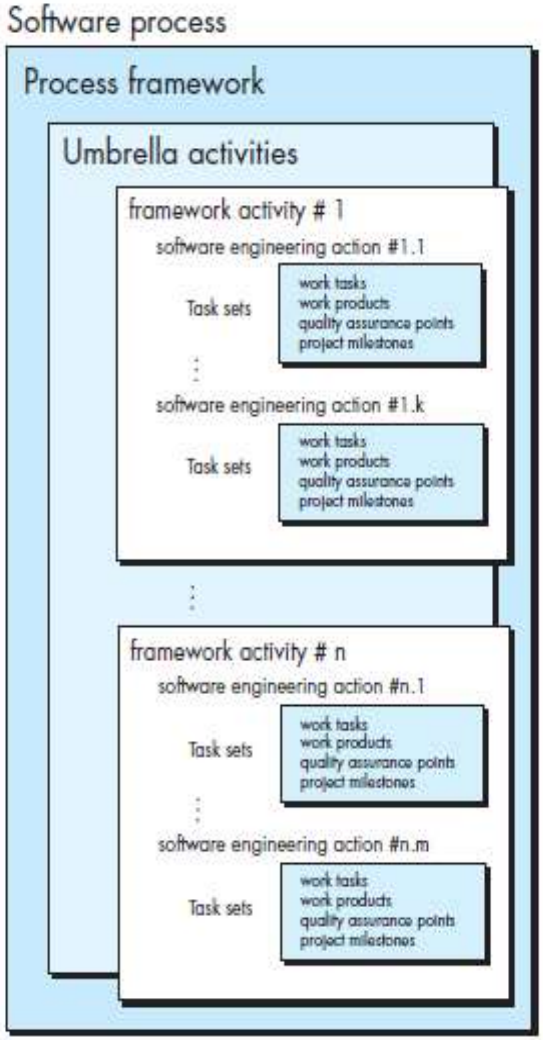
- 3.1 A Generic Process Model
- 3.2 Defining a Framework Activity
- 3.3 Process Assessment and Improvement

3.1 A GENERIC PROCESS MODEL

- The hierarchy of technical work within the software process is activities, encompassing actions, populated by tasks.

Software Process Framework

FIGURE 3.1
A software process framework



****Slide Content is taken from "Software Engineering, A Practitioner's Approach", by Roger S Pressman & Bruce R Maximum (8E)****

Toomula Srilatha,
MSc, MTech SET (PhD)
Assistant Professor, REVRR Women's College, Hyd

3.2 DEFINING A FRAMEWORK ACTIVITY

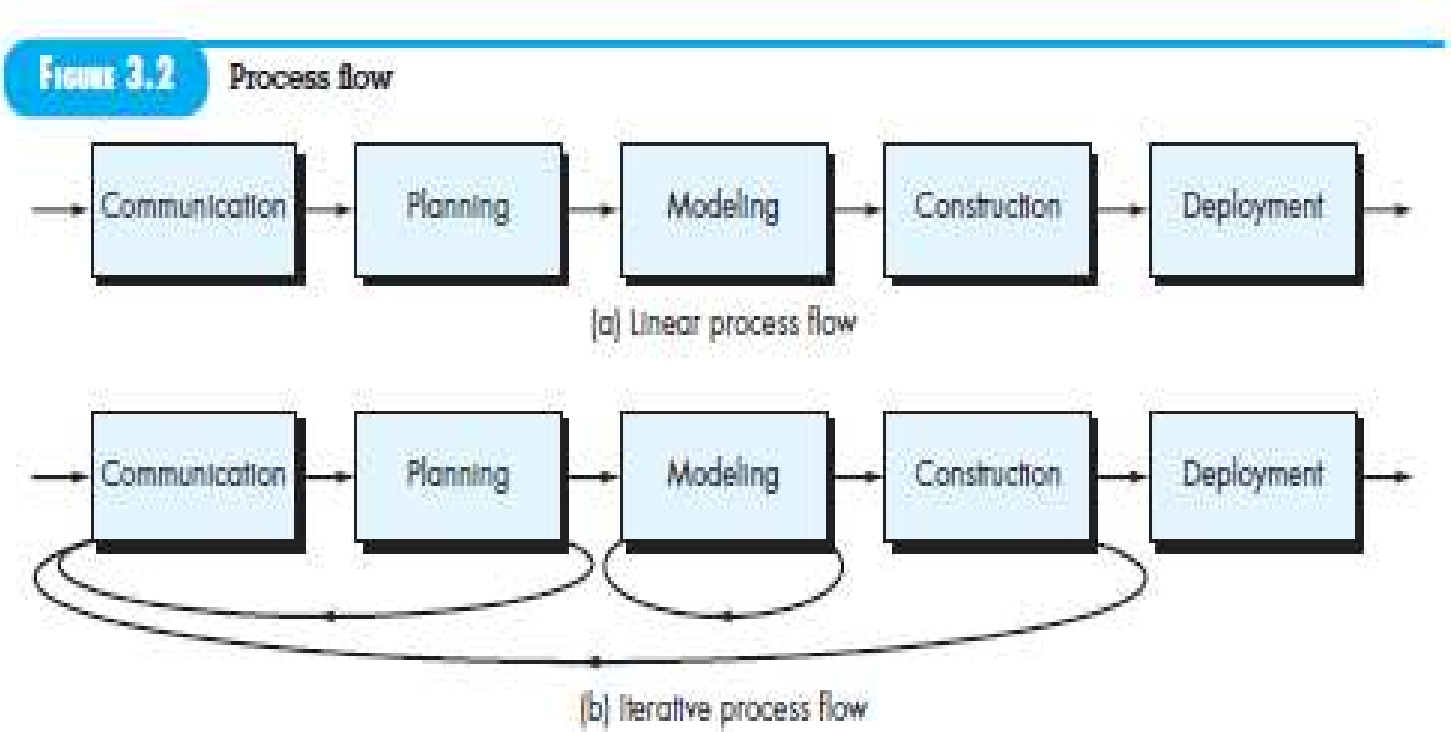
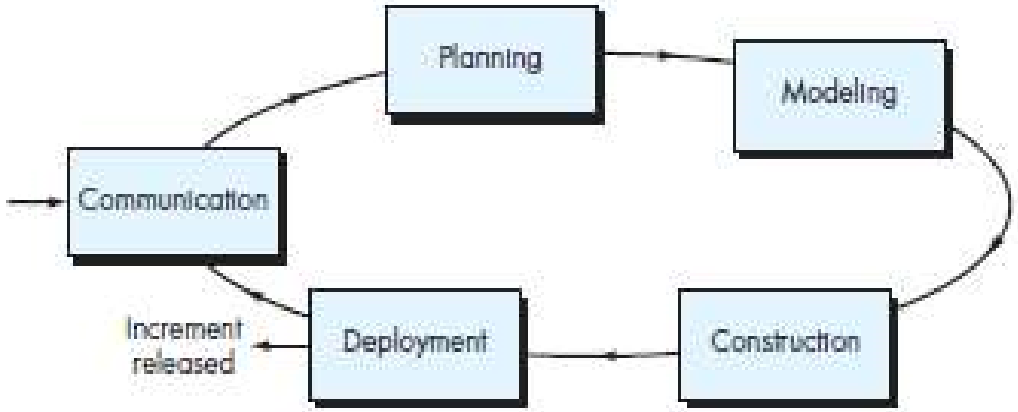


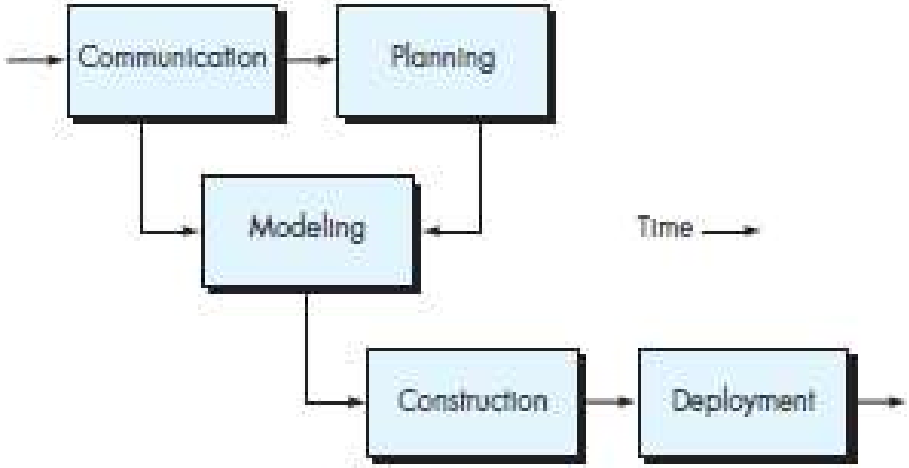
FIGURE 3.2 Process flow

Toomula Srilatha,
MSc, MTech SET (PhD)
Assistant Professor, RBVRR Women's College, Hyd

3.2 DEFINING A FRAMEWORK ACTIVITY



(c) Evolutionary process flow



(d) Parallel process flow

3.3 IDENTIFYING A TASK SET

- Each software engineering action (e.g., *elicitation,* an action associated with the **communication activity) can be represented by a** number of different *task sets.*
- ***Task set*** *is a collection of software engineering work* tasks, related work products, quality assurance points, and project milestones.

INFO

Task Set

A task set defines the actual work to be done to accomplish the objectives of a software engineering action. For example, *elicitation* (more commonly called "requirements gathering") is an important software engineering action that occurs during the **communication** activity. The goal of requirements gathering is to understand what various stakeholders want from the software that is to be built.

For a small, relatively simple project, the task set for requirements gathering might look like this:

1. Make a list of stakeholders for the project.
2. Invite all stakeholders to an informal meeting.
3. Ask each stakeholder to make a list of features and functions required.
4. Discuss requirements and build a final list.
5. Prioritize requirements.
6. Note areas of uncertainty.

For a larger, more complex software project, a different task set would be required. It might encompass the following work tasks:

1. Make a list of stakeholders for the project.
2. Interview each stakeholder separately to determine overall wants and needs.
3. Build a preliminary list of functions and features based on stakeholder input.
4. Schedule a series of facilitated application specification meetings.
5. Conduct meetings.
6. Produce informal user scenarios as part of each meeting.
7. Refine user scenarios based on stakeholder feedback.
8. Build a revised list of stakeholder requirements.
9. Use quality function deployment techniques to prioritize requirements.
10. Package requirements so that they can be delivered incrementally.
11. Note constraints and restrictions that will be placed on the system.
12. Discuss methods for validating the system.

Both of these task sets achieve "requirements gathering," but they are quite different in their depth and formality. The software team chooses the task set that will allow it to achieve the goal of each action and still maintain quality and agility.

3.4 PROCESS PATTERNS

- A *process pattern describes a process-related problem that is encountered during* software engineering work, identifies the environment in which the problem has been encountered, and suggests one or more proven solutions to the problem.
- Stated in more general terms, a process pattern provides you with a template-a consistent method for describing problem solutions within the context of the software process.
- By combining patterns, a software team can solve problems and construct a process that best meets the needs of a project.

3.4 PROCESS PATTERNS

- Ambler has proposed a template for describing a process pattern:
- **Pattern Name. The pattern is given a meaningful name describing it**
- **Type. The pattern type is specified.**
- **Ambler suggests three types of Patterns:**
- **1. Stage pattern —defines a problem associated with a framework**
- An example of a stage pattern might be **Establishing Communication .**

3.4 PROCESS PATTERNS

- **2. Task pattern — defines a problem associated with a software engineering** action or work task and relevant to successful software engineering practice (e.g., **Requirements Gathering is a task pattern).**
- **3. Phase pattern — define the sequence of framework activities that occurs** within the process, even when the overall flow of activities is iterative in nature.
- An example of a phase pattern might be **Spiral Model or Prototyping .**

INFO

An Example Process Pattern

The following abbreviated process pattern describes an approach that may be applicable when stakeholders have a general idea of what must be done but are unsure of specific software requirements.

Pattern Name. **RequirementsUnclear**

Intent. This pattern describes an approach for building a model (a prototype) that can be assessed iteratively by stakeholders in an effort to identify or solidify software requirements.

Type. Phase pattern.

Initial Context. The following conditions must be met prior to the initiation of this pattern: (1) stakeholders have been identified; (2) a mode of communication between stakeholders and the software team has been established; (3) the overriding software problem to be solved has been identified by stakeholders; (4) an initial understanding of project scope, basic business requirements, and project constraints has been developed.

Problem. Requirements are hazy or nonexistent, yet there is clear recognition that there is a problem to be solved, and the problem must be addressed with a software solution. Stakeholders are unsure of what they want; that is, they cannot describe software requirements in any detail.

Solution. A description of the prototyping process would be presented here and is described later in Section 4.1.3.

Resulting Context. A software prototype that identifies basic requirements (e.g., modes of interaction, computational features, processing functions) is approved by stakeholders. Following this, (1) the prototype may evolve through a series of increments to become the production software or (2) the prototype may be discarded and the production software built using some other process pattern.

Related Patterns. The following patterns are related to this pattern: **CustomerCommunication, IterativeDesign, IterativeDevelopment, CustomerAssessment, RequirementExtraction.**

Known Uses and Examples. Prototyping is recommended when requirements are uncertain.

3.5 PROCESS ASSESSMENT AND IMPROVEMENT

- Assessment attempts to understand the current state of the software process with the intent of improving it.
- A number of different approaches to software process assessment and improvement have been proposed over the past few decades:
  - **Standard CMMI Assessment Method for Process Improvement (SCAMPI)**
  - **CMM-Based Appraisal for Internal Process Improvement (CBA IPI)**
  - **SPICE (ISO/IEC15504)**
  - **ISO 9001:2000 for Software**

Chapter – 4
Process Models

- 4.1 Prescriptive Process Models
  - 4.1.1 The Waterfall Model
  - 4.1.2 Incremental Process Model
  - 4.1.3 Evolutionary Process Models
  - 4.2.4 Concurrent Models
  - 4.2.5 A Final Word on Evolutionary Processes
- 4.2 Specialized Process Models
  - 4.2.1 Component-Based Development
  - 4.2.2 The Formal Methods Model
  - 4.2.3 Aspect-Oriented Software Development
- 4.3 The Unified Process
  - 4.3.1 A Brief History
  - 4.3.2 Phases of the Unified Process
- 4.4 Personal and Team Process Models
  - 4.4.1 Personal Software Process
  - 4.4.2 Team Software Process

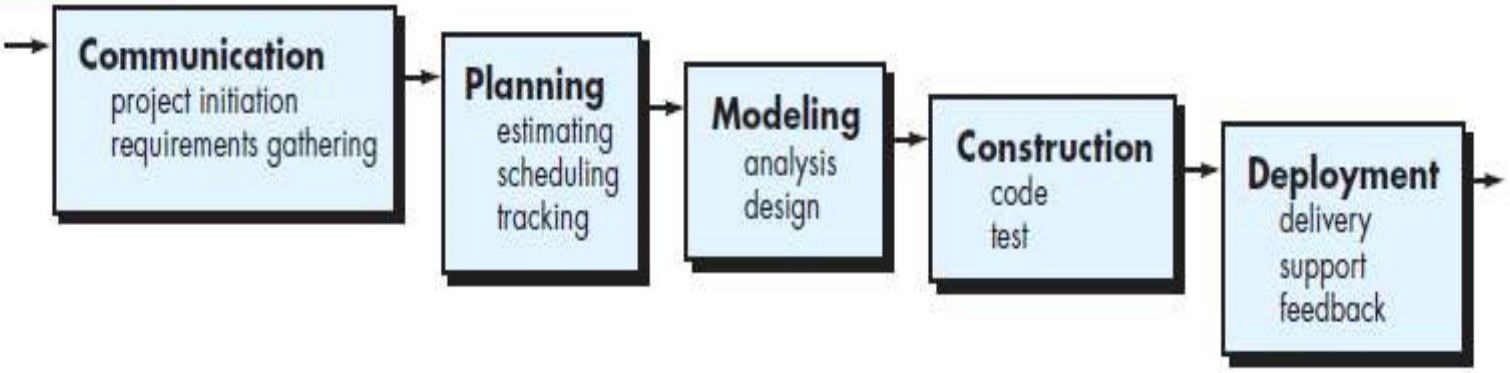
4.1 PRESCRIPTIVE PROCESS MODELS

- A *prescriptive process model strives for structure and order in software development.*
- Activities and tasks occur sequentially with defined guidelines for progress.

4.1.1 The Waterfall Model

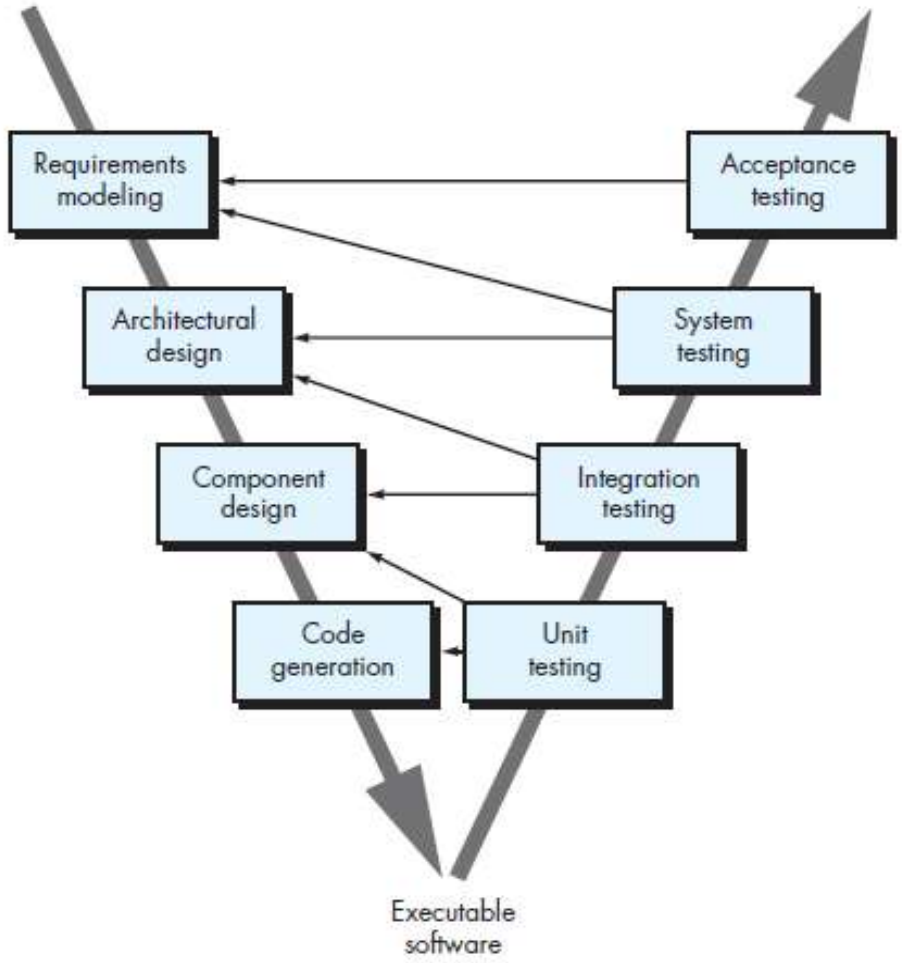
- The *waterfall model, sometimes called the* ***classic life cycle*** *, suggests a systematic,* sequential approach to software development that begins with customer specification of requirements and progresses through planning, modeling, construction, and deployment, culminating in ongoing support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

FIGURE 4.1 The waterfall model



The V-model

FIGURE 4.2
The V-model

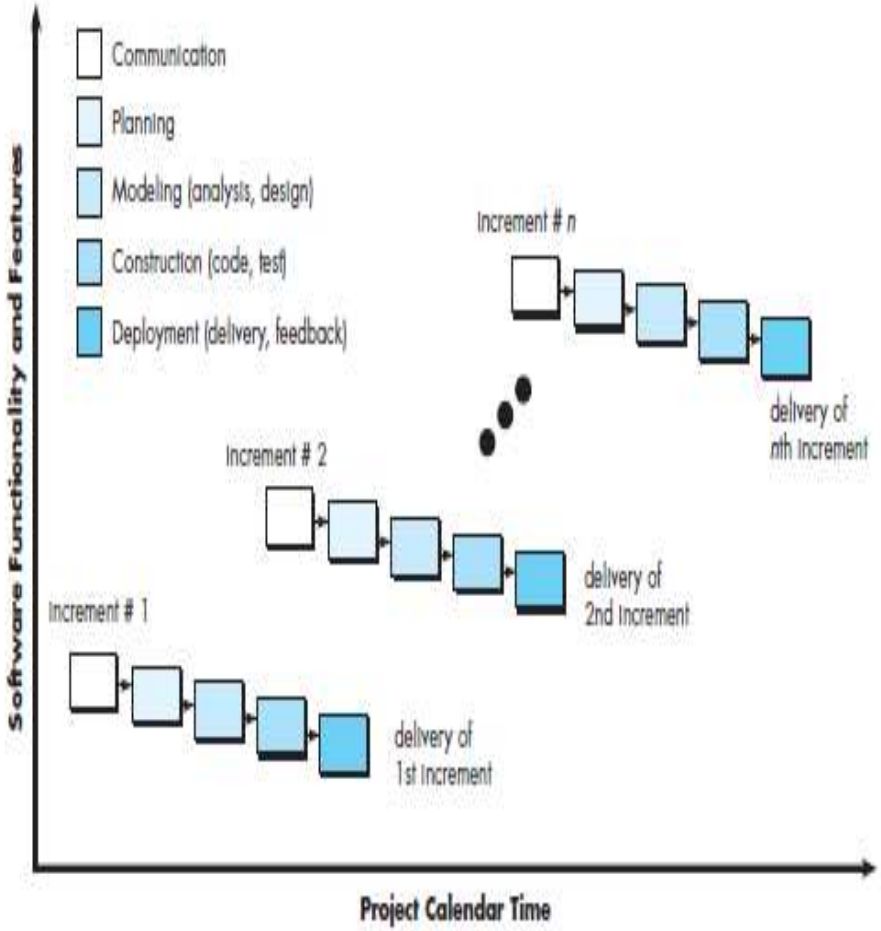


- "Too often, software work follows the first law of bicycling:
- No matter where you're going, it's uphill and against the wind."

4.1.2 Incremental Process Models

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

FIGURE 4.3
The incremental model



4.1.3 Evolutionary Process Models

- Evolutionary process models produce an increasingly more complete version of the software with each iteration.
- Software, like all complex systems, evolves over a period of time.
- Business and product requirements often change as development proceeds
- A set of core product or system requirements is well understood
- A process model is required that has been explicitly designed to accommodate a product that grows and changes.
- Evolutionary models are iterative.
- They are characterized in a manner that enables you to develop increasingly more complete versions of the software.
- In the next slides, we present two common evolutionary process models.

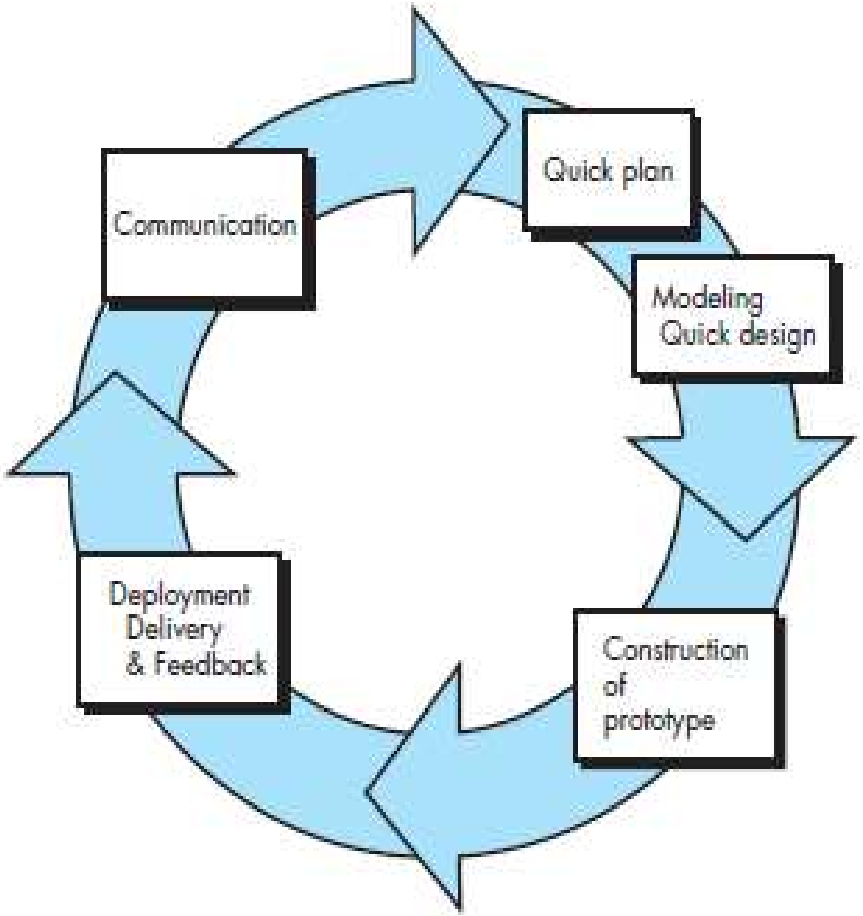
Prototyping

- 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 situations, **a *prototyping paradigm*** *may* offer the best approach.
- The prototyping paradigm begins with communication.
- 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 then modeling occurs.

Prototyping

FIGURE 4.4

The prototyping paradigm



Spiral model

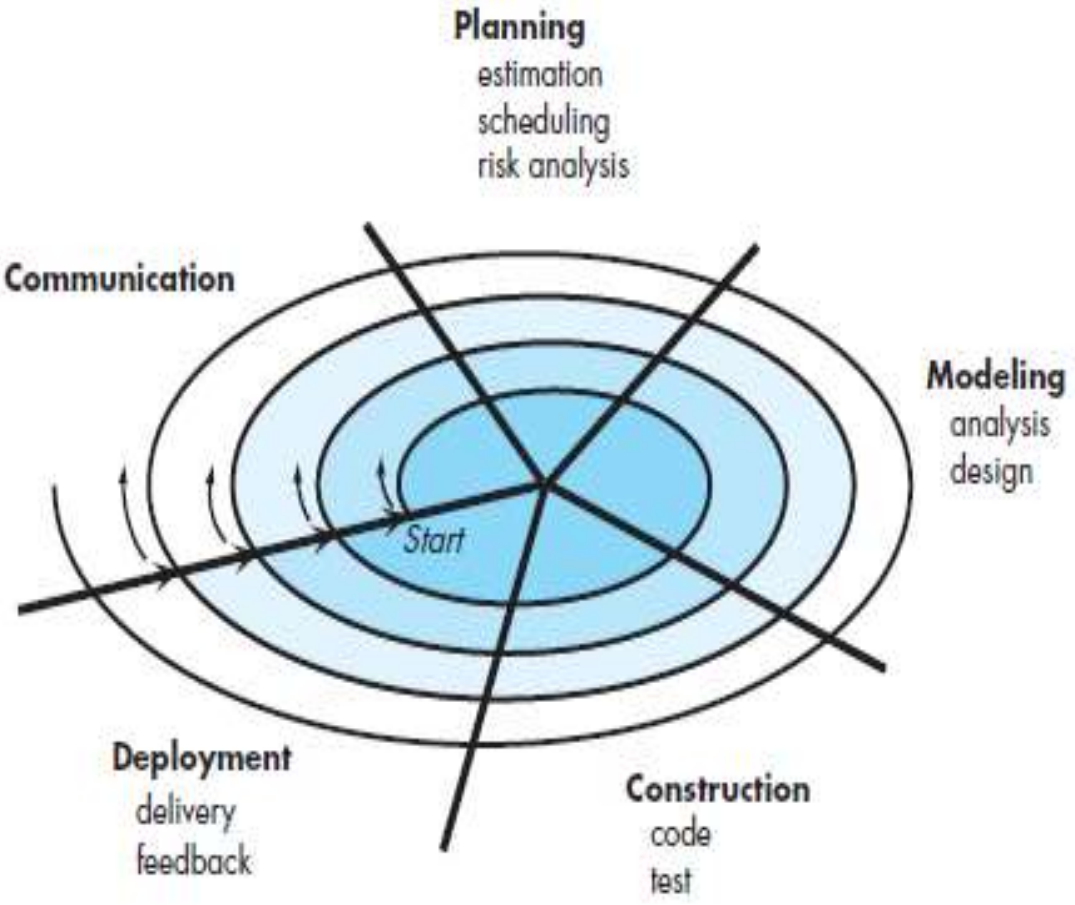
- A spiral model is divided into a set of framework activities defined by the software engineering team.
- For illustrative purposes, we use the generic framework activities discussed earlier.
- Each of the framework activities represent one segment of the spiral path illustrated in Figure 4.5

.

Spiral Model

FIGURE 4.5

A typical spiral model



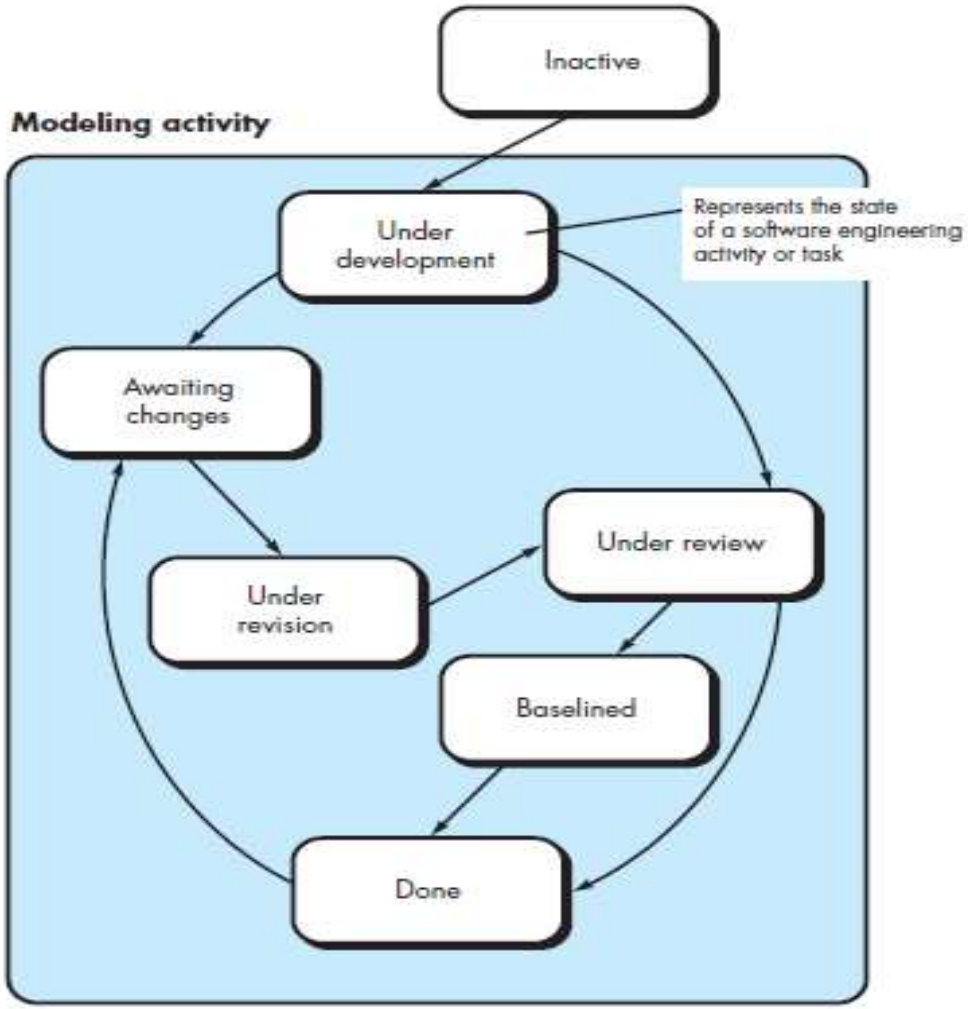
4.1.4 Concurrent Models

- The ***concurrent development model***, *sometimes called concurrent engineering,* allows a software team to represent iterative and concurrent elements of any of the process models.
- For example, the modeling activity defined for the spiral model is accomplished by invoking one or more of the following:
- software engineering actions: prototyping, analysis, and design.
- Figure 4.6 provides an example of the concurrent modeling approach.
- An activity— **modeling —may be in any one of the states 7 noted at any given time.**
- 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.

4.1.4 Concurrent Models

FIGURE 4.6

One element of the concurrent process model



4.2 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.

4.2.1 Component-Based Development

- **Commercial off-the-shelf (COTS)** software components or 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 with **Iterative** approach.
- **Candidate components** can be designed as either conventional software modules or object-oriented classes or packages of classes.

4.2.1 Component-Based Development

- The component-based development model incorporates the following steps :

1. Available component-based products are researched and evaluated for

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.

Toomula Srilatha,
MSc, MTech SET (PhD)
Assistant Professor, RBVRR Women's College, Hyd

4.2.2 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 ***clean room software engineering****, is currently applied by some software development* organizations.

4.2.2 The Formal Methods Model

- Formal methods provide a mechanism for eliminating many of the problems that are difficult to overcome.
- Ambiguity, incompleteness, and inconsistency can be discovered and corrected through the application of mathematical analysis.
- When formal methods are used during design, they serve as a basis for program verification.
- Defect-free software.
- The development of formal models is quiet time consuming and expensive.

Toomula Srilatha,
MSc, MTech SET (PhD)
Assistant Professor, RBVRR Women's College, Hyd

4.2.3 Aspect-Oriented Software Development

- 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)** or **aspect-oriented component engineering (AOCE).**
- AOCE 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".

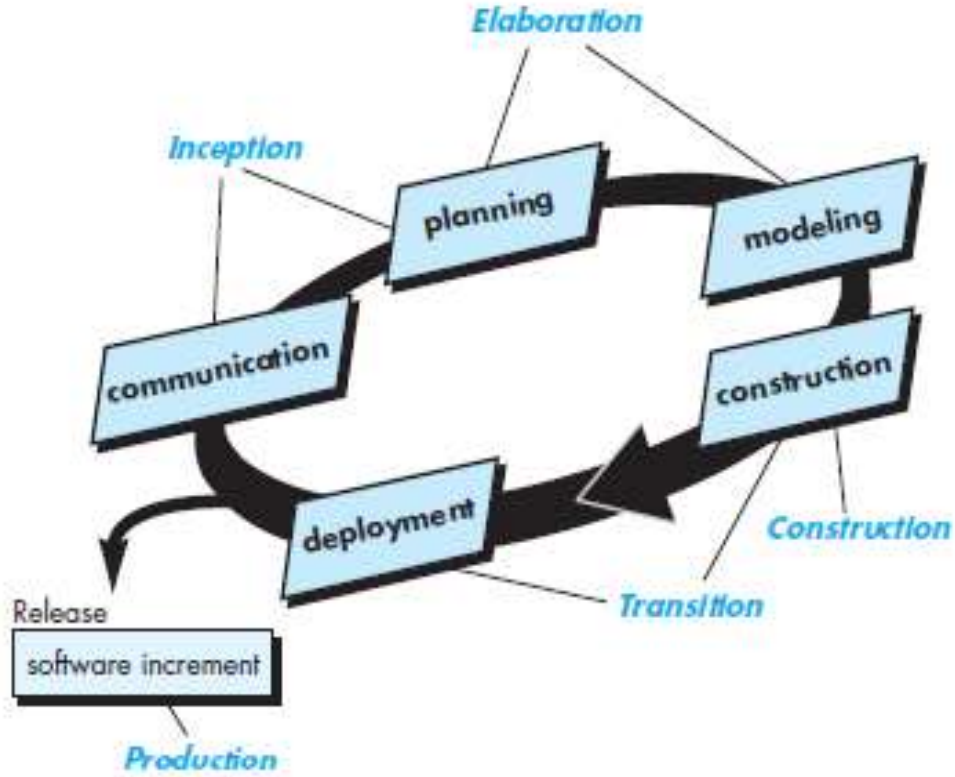
4.3 THE UNIFIED PROCESS

- During the early 1990s James Rumbaugh, Grady Booch , and Ivar Jacobson began working on a "unified method".
- Unified Method combines the **best features** of each of object-oriented analysis and design methods.
- This process also adopted additional features proposed by the experts 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.

4.3.2 Phases of the Unified Process 14

- The inception phase of the UP encompasses both customer communication and planning activities.

FIGURE 4.7
The Unified Process



4.3.2 Phases of the Unified Process 14

- The **elaboration phase** encompasses the communication and modeling activities of the generic process model
- The **construction phase** of the UP is identical to the construction activity defined for the generic software process.
- 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.
- 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.

4.4.1 Personal Software Process

The **PSP** model defines **five** framework activities:

Planning. This activity isolates requirements and develops both size and resource estimates.

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 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 important tasks and work results.

Postmortem. Using the measures and metrics collected, the effectiveness of the process is determined. Measures and metrics should provide guidance for modifying the process to improve its effectiveness.

4.4.2 Team Software Process

- **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 CMMI level 5 behavior normal and expected.
  - Provide improvement guidance to high-maturity organizations.
  - Facilitate university teaching of industrial-grade team skills.

4.4.2 Team Software Process

- TSP defines the following framework activities: project launch,
  - high-level design
  - implementation
  - integration and test and
  - postmortem

Chapter – 5
Agile Development

- 5.1 What is Agility?
- 5.2 Agility and Cost of Change
- 5.3 What is Agile Process?
  - 5.3.1 Agility Principles
  - 5.3.2 The politics of Agile Development
- 5.4 Extreme Programming
  - 5.4.1 The XP Process
  - 5.4.2 Industrial XP

5.1 WHAT IS AGILITY ?

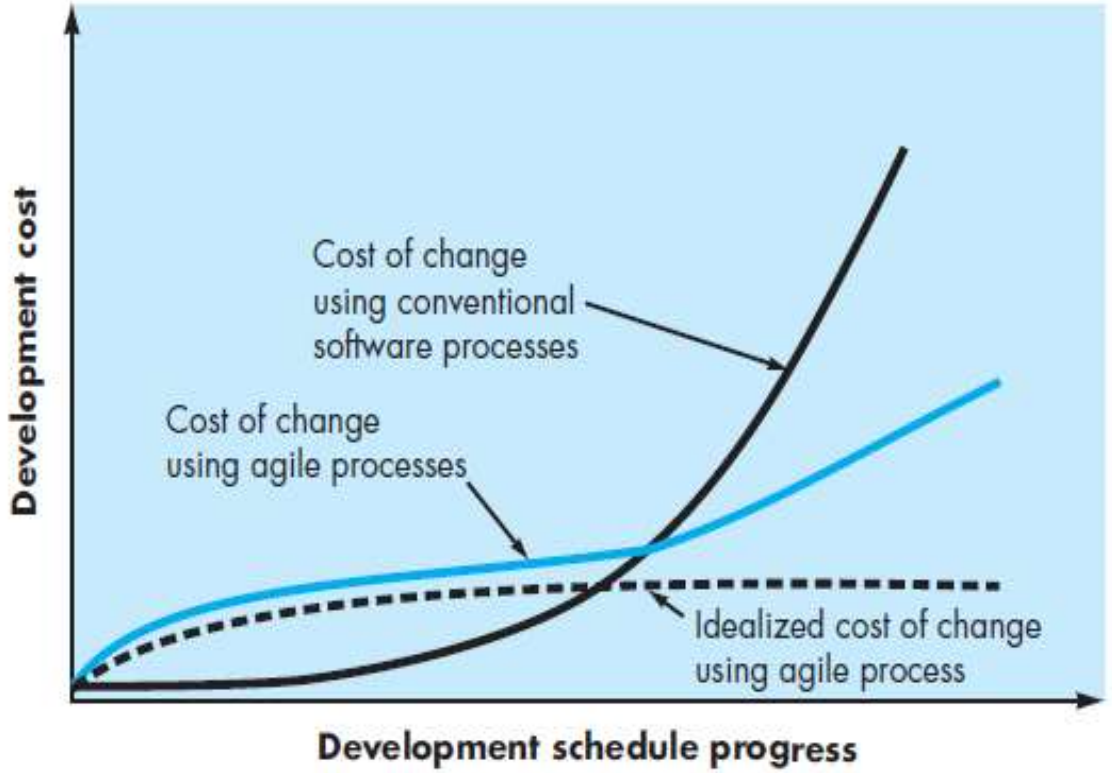
- **Agility** has become today's buzzword when describing a modern software process.
- Everyone is **agile**.
- An agile team is a nimble team able to appropriately respond to Changes.
- **Agility** can be applied to any software process.
- Agility supports incremental delivery
- Strategy that gets working software to the customer as rapidly as feasible.

Toomula Srilatha,
MSc, MTech SET (PhD)
Assistant Professor, REVRR Women's College, Hyd

5.2 AGILITY AND THE COST OF CHANGE

FIGURE 5.1

Change costs as a function of time in development



5.3 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.
- 2. For many types of software, design and construction are interleaved.
- 3. Analysis, design, construction, and testing are not as predictable (from a planning point of view) as we might like.

5.3.1 Agility Principles

The Agile Alliance defines 12 agility principles for those who want to achieve agility:

1. **Our highest priority is to satisfy the customer through early and continuous** delivery of valuable software.
2. **Welcome changing requirements.**
3. **Deliver working software frequently** with the shorter timescale.
4. **Business people and developers must work together daily throughout the** project.
5. **Build projects around motivated individuals.**
6. **The most efficient and effective method of conveying info is 1-to-1** conversation.
7. **Working software is the primary measure of progress.**
8. **Agile processes promote sustainable development.**
9. **Continuous attention to technical excellence and good design enhances** agility.
10. **Simplicity—the art of maximizing the amount of work not done—is essential.**
11. **Best architectures, requirements & designs emerge from self-organizing** teams.
12. **At regular intervals, the team reflects on how to become more effective**.

5.3.2 The Politics of Agile Development

- There has been considerable debate about the benefits and applicability of agile software development as opposed to more conventional software engineering processes.
- Jim Highsmith states the extremes when he characterizes the feeling of the pro-agility camp.

5.4 EXTREME PROGRAMMING

- In order to illustrate an agile process in a bit more detail, there is an overview of ***Extreme Programming (XP), the most widely used approach to*** **agile software development.**
- Although early work on the ideas and methods associated with XP occurred during the late 1980s, the seminal work on the subject has been written by **Kent Beck.**
- A variant of XP, called ***Industrial XP*** **(IXP),** refines XP and targets the agile process specifically for use within large organizations.

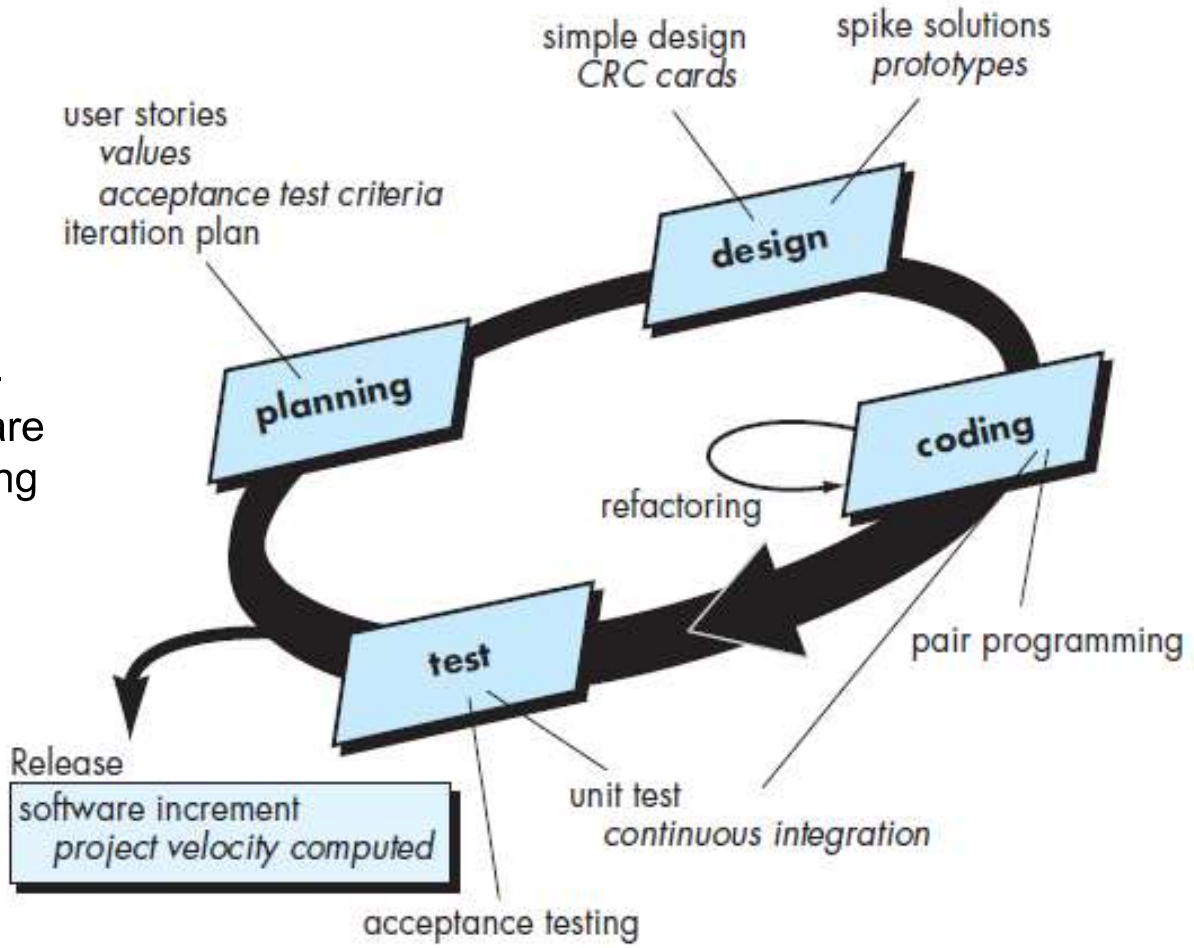
5.4.1 The XP Process

- **Extreme Programming** uses an **object-oriented approach** as its preferred development paradigm and
- encompasses a **set of rules** and **practices** that occur within the context of **four** framework activities: planning, design, coding, and testing.
- Figure 5.2 (next) illustrates the XP process
- Key XP activities are summarized in the paragraphs that follow.
- **Planning.** This activity *begins with* ***requirements gathering*** *that enables the technical members* of the XP team to understand the business context for the software and
- to get a broad feel for required output, major features and functionality

5.4.1 The XP Process

FIGURE 5.2

The Extreme Programming process



Class-Responsibility-Collaboration cards are tools used in designing object-oriented software

5.4.1 The XP Process

Design - A simple design is always preferred over a more complex representation.

Coding - A key concept during the coding activity is *pair programming* .

Testing –

- The **unit tests** that are created should be implemented using a framework that enables them to be automated
- **Regression testing** strategy is applied whenever code is modified
- As the individual unit tests are organized into a "universal testing suite" integration and validation testing of the system can occur on a daily basis.
- **XP *acceptance tests*** , *also called* ***customer tests****, are specified by the customer* and focus on overall system features and functionality that are visible and reviewable by the customer.

5.4.2 Industrial XP

- Joshua Kerievsky describes Industrial Extreme Programming (IXP) as: "IXP is an organic evolution of XP".
- **IXP** incorporates **six** new practices that are designed to help ensure that an XP project works successfully. They are
  1. Readiness assessment.
  2. Project community.
  3. Project chartering.
  4. Test-driven management.
  5. Retrospectives. An IXP team conducts a specialized technical review after a software increment is delivered, which is Called a retrospective.
  6. Continuous learning. The IXP team is encouraged to learn new methods and techniques that can lead to a higher-quality product.

Chapter – 6
Human Aspects of Software Engineering

- 6.1 Characteristics of a Software Engineer
- 6.2 The Psychology of Software Engineering
- 6.3 The Software Team
- 6.4 Team Structures
- 6.7 Software Engineering using the Cloud
- 6.9 Global Teams

Human Aspects of Software Engineering

- Software engineering has an abundance of techniques, tools, and methods designed to improve both the software development process and the final product.
- Technical improvements continue to emerge and yield encouraging results.

6.1 CHARACTERISTICS OF A SOFTWARE ENGINEER

- An effective software engineer has a **sense of individual responsibility**
- An effective software engineer has an **acute awareness of the needs of other members** of his team.
- An effective software engineer is **honest**
- An effective software engineer exhibits **resilience under pressure.**
- An effective software engineer has a **heightened sense of fairness.**
- An effective software engineer **exhibits attention to detail**.
- Finally, an effective software engineer is **pragmatic**

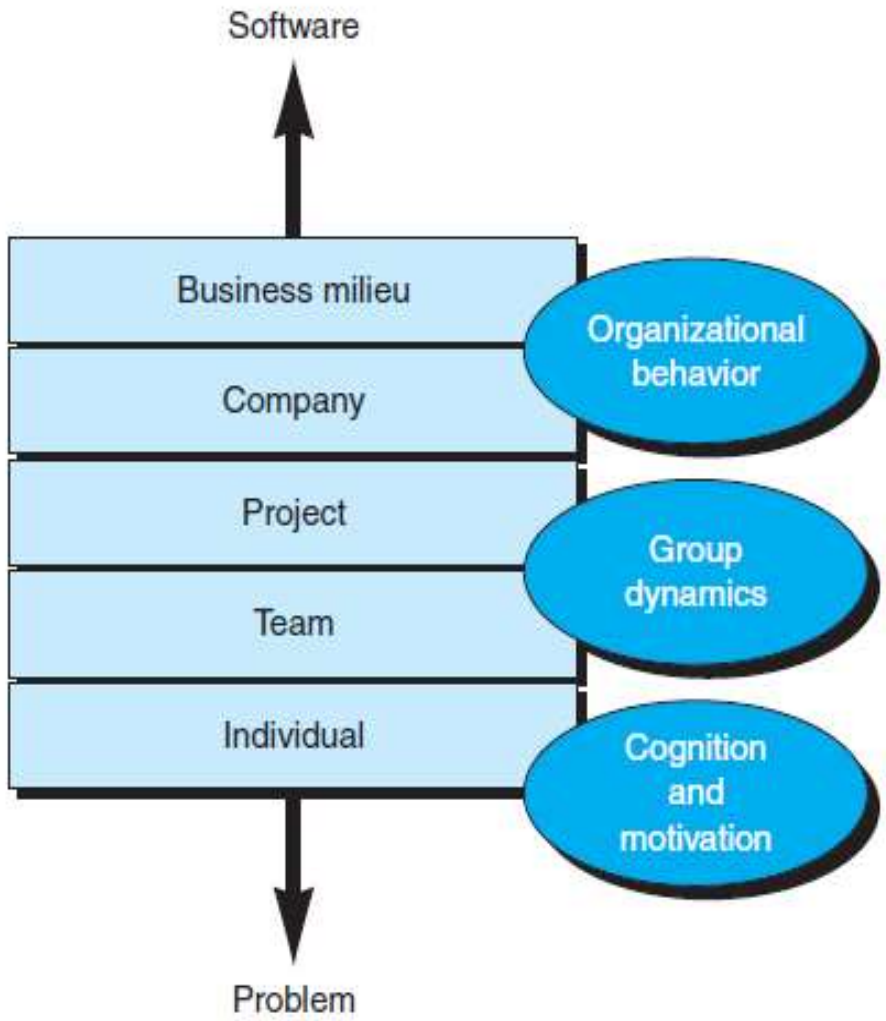
6.2 THE PSYCHOLOGY OF SOFTWARE ENGINEERING

- Software engineering psychology focuses on recognition of the problem to be solved, problem-solving skills required to solve it, and the motivation to complete the solution within the constraints.
- At the team and project levels, group dynamics becomes the dominating factor.
- Here, team structure and social factors govern success.
- Group communication, collaboration, and coordination are as important as the skills of an individual team member.

6.2 THE PSYCHOLOGY OF SOFTWARE ENGINEERING

FIGURE 6.1

A layers behavioral model for software engineering (adapted from [Cur90])



6.2 THE PSYCHOLOGY OF SOFTWARE ENGINEERING

The following roles may be assigned explicitly or can evolve naturally:

- **Ambassador** —represents the team to outside constituencies with the intent of negotiating time and resources and gaining feedback from stakeholders.
- **Scout** —crosses the team's boundary to collect organizational information. "Scouting can include scanning about external markets, searching for new technologies & identifying relevant activities outside of the team"
- **Guard** —protects access to the team's work products and other information artifacts.
- **Sentry** —controls the fl ow of information that stakeholders and others send to the team.
- **Coordinator** —focuses on communicating horizontally across the team and within the organization.

6.3 THE SOFTWARE TEAM

- DeMarco and Lister contend that members of jelled teams are significantly more productive and more motivated than average.
- They share a common goal, a common culture, and in many cases, a "**sense of eliteness**" that makes them unique.
- An effective team should foster a ***sense of trust***.
- The team should encourage a ***sense of improvement***

6.3 THE SOFTWARE TEAM

She defines five factors that "foster a potentially toxic team environment":

- (1) a frenzied work atmosphere,
- (2) high frustration that causes friction among team members,
- (3) a "fragmented or poorly coordinated" software process,
- (4) an unclear definition of roles on the software team &
- (5) "continuous and repeated exposure to failure."

Toomula Srilatha,
MSc, MTech SET (PhD)
Assistant Professor, RBVRR Women's College, Hyd

6.4 TEAM STRUCTURES

- The “best” team structure depends on the management style of your organization, the number of people who will populate the team and their skill levels, and the overall problem difficulty.
- Mantei describes a number of project factors that should be considered when planning the structure of software engineering teams:

  (1) difficulty of the problem to be solved,

  (2) “size” of the resultant program in lines of code or function points,

  (3) time that the team will stay together,

  (4) degree to which the problem can be modularized,

  (5) required quality and reliability of the system to be built,

  (6) rigidity of the delivery date, and

  (7) degree of sociability required for the project.

6.4 TEAM STRUCTURES

Constantine suggests four "organizational paradigms" for software engineering teams:

1. **A *closed paradigm*** *structures a team along a traditional hierarchy of authority.*
2. **A *random paradigm*** *structures a team loosely and depends on individual* initiative of the team members.
3. **An *open paradigm*** *attempts to structure a team in a manner that achieves* some of the controls associated with the closed paradigm Open paradigm team structures are well suited to the solution of complex problems but may not perform as efficiently as other teams.
4. **A *synchronous paradigm*** *relies on the compartmentalization of a* problem and organizes team members to work on pieces of the problem with little active communication among themselves.

6.7 SOFTWARE ENGINEERING USING THE CLOUD

- Cloud computing provides a mechanism for access to all software engineering work products, artifacts, and project-related information.
- It runs everywhere and removes the device dependency that was once a constraint for many software projects.
- It allows members of a software team to conduct platform-independent & low-risk trials of new software tools
- It provides the potential for improved approaches to content and configuration management
- Software engineering information developed by one team member can be instantly available to all team members, regardless of the platform others are using.
- Information dispersion speeds up and broadens dramatically.
- As Gardner states, one of the key benefits of the cloud is its ability to enhance the "social and collaborative aspects of software development."

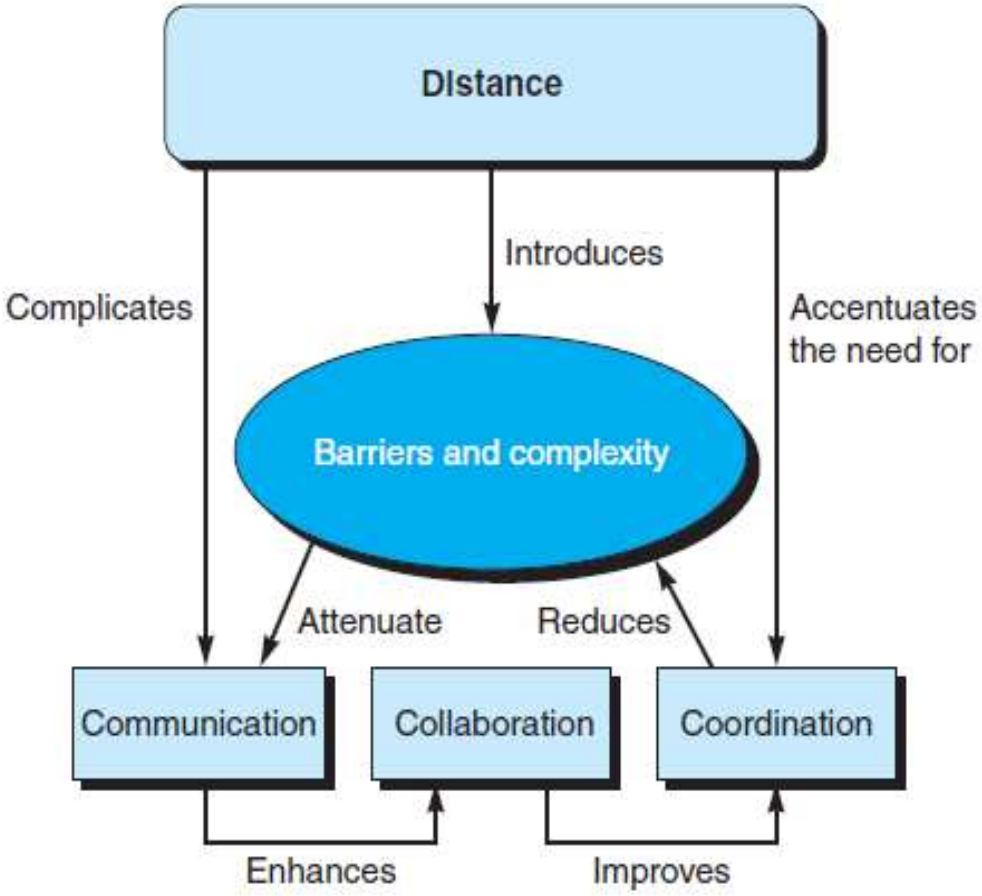
6.9 GLOBAL TEAMS

- In the software domain, globalization implies more than the transfer of goods and services across international boundaries.
- For the past few decades, an increasing number of major software products have been built by software teams that are often located in different countries.
- These global software development (GSD) teams have many of the characteristics of a conventional software team
- Decision making on all software teams is complicated by four factors :
  - Complexity of the problem
  - Uncertainty and risk associated with the decision
  - The law of unintended consequences
  - Different views of the problem that lead to different conclusions about the way forward.

6.9 GLOBAL TEAMS

FIGURE 6.2

Factors affecting a GSD team (adapted from [Cas06])



Summary

Chapter – 1: The Nature of Software

Chapter – 2: Software Engineering

Chapter – 3: Software Process Structure

Chapter – 4: Process Models

Chapter – 5: Agile Development

Chapter – 6: Human Aspects of Software Engineering

Toomula Srilatha,
MSc, MTech SET (PhD)
Assistant Professor, RBVRR Women's College, Hyd

END OF UNIT-1