

# Software Engineering

2UCC501

# MODULE 1: The Product and the Process (08)

## CO 1: Understand the software development process and Estimate different types of resources for the given project

- 1.1 Software Development Life Cycle models: Waterfall, RAD, Spiral , Agile process
- 1.2 Understanding software process, Process metric, CMM levels
- 1.3 Planning & Estimation: Product metrics estimations- LOC, FP, COCOMO models
- 1.4 Project Management Activities: Planning , Scheduling & Tracking

# Software

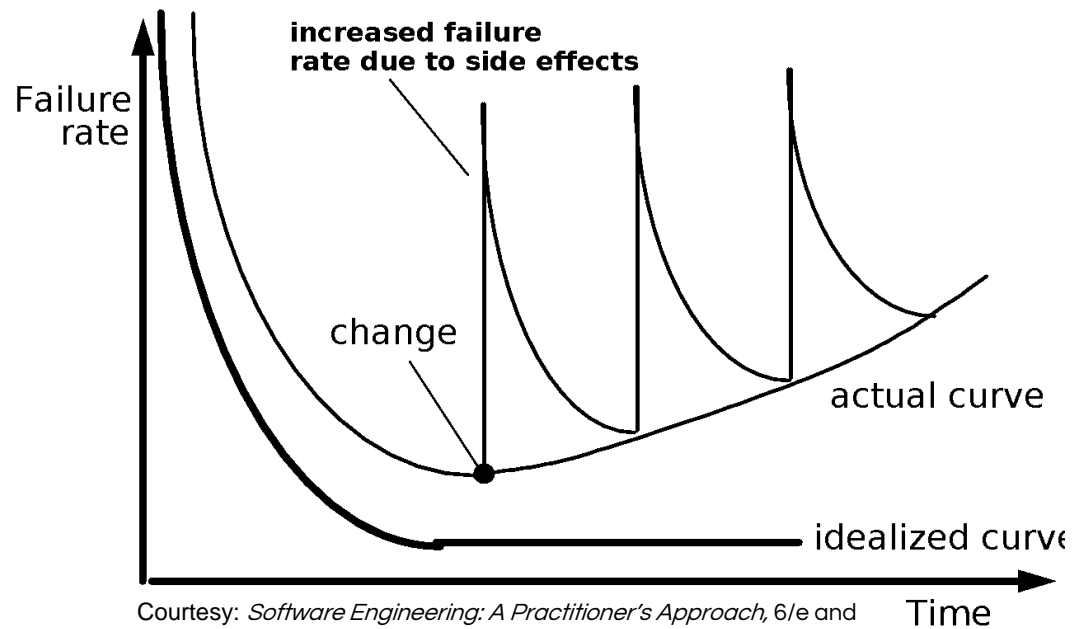
- Software: a dual role product
  - Product: Supports or directly provides system functionality
  - Produces, manages, acquires, modifies, displays, or transmits information
  - Tool: Helps build other software Delivers computing potential

# Software includes:

- Program/ Product :
  - Operating System
  - API/ Library Functions/ Apps
  - Databases
- Documents:
  - Problem Definition
  - Method to solve / approach
  - Code (Reference manual)
  - Users Manual
  - Test reports

# Software

- Manufactured / Engineered
- Does not wear & tear



Courtesy: *Software Engineering: A Practitioner's Approach*, 6/e and are provided with permission by R.S. Pressman & Associates, Inc., copyright © 1996, 2001, 2005

- Complex

# Types / Classification of Software

- System Software:
  - Operating System
  - Device Drivers
- Application Software:
  - Games
  - Apps
- Generic Software:
- Customized Software:

# Definition of Software Engineering

- **Software**
  - Document / Code
- **Engineering**
  - Systematic Development approach
  - Optimum utilization of resources
- **IEEE** defines **software engineering** as: (1) The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of **software**; that is, the application of **engineering** to **software**. (2) Study of approaches as in (1).
- Fritz Bauer defined Software engineering as the “*establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.*”

# Definition of Software Engineering

- Software Engineering is the branch of engineering which deals with the establishment and use of sound engineering principles to obtain economical Software that is reliable and works efficiently on real machines.



# Fill in the blank:

- Define Software Engineering:

---

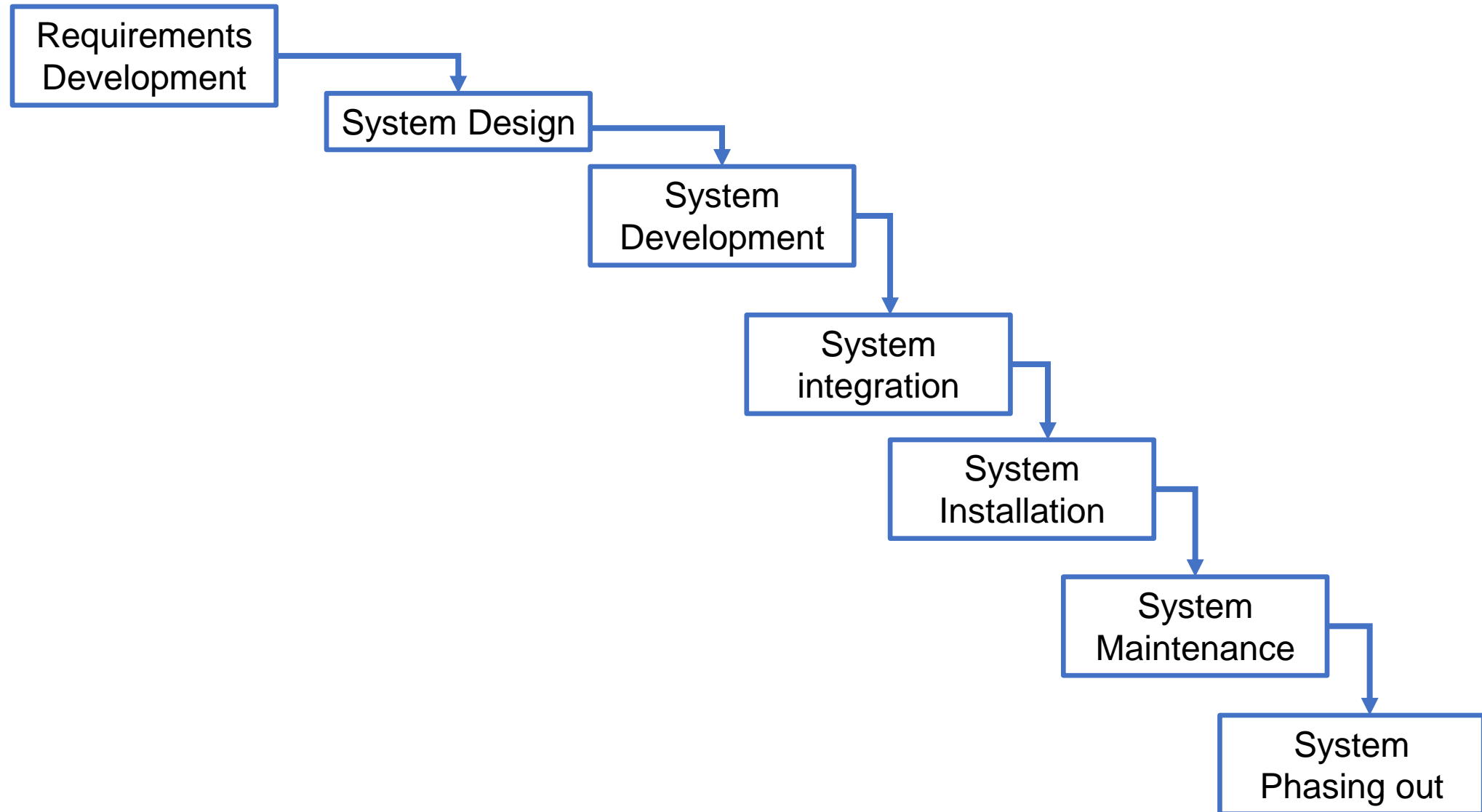
- Software Development is an Art or a Science. Your comments:

---

# System Engineering & Software Engineering

- System: Set of parts interconnected in such a way to achieve common objectives
- System Engineering focuses on a specific domain such as business or product
- Software: Program, Installation manual, Documentation

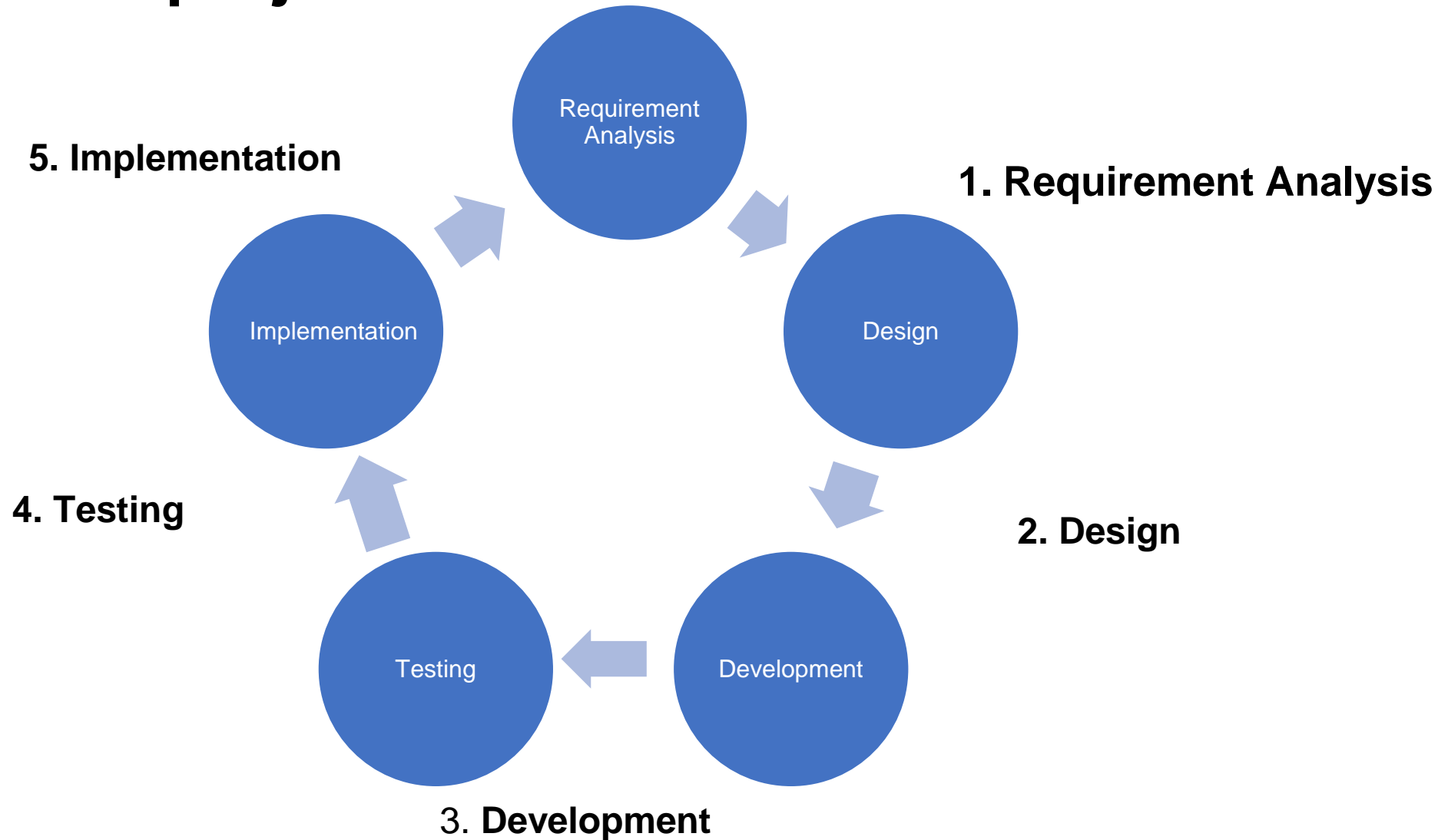
# Processes of System Engineering



# 4 Ps Related to Software Engineering (SE)

- **Product**
  - The deliverables
- **Process**
  - Development
  - Testing
- **Project**
  - Systematic approach with specific characteristics
- **People**
  - Designers, Analysts, Developers, Users

# Activities performed in life cycle of a Software Development projects



# Steps involved in Software Development

1. Identification of Problem
2. Problem Definition
3. Exploring approach(s) to solve the problem
4. Penning down/ finalizing the approach; listing the steps
5. Development
6. Testing
7. Deployment
8. Maintenance

These steps are iterative and some of them run parallel

# Generic Software Development Process Models

1. Code & Fix Model

2. Waterfall model

2016 Q1

2017 Q1

3. Rapid Application Development (RAD)

4. Spiral

5. Iterative

6. Evolutionary

7. Agile

# Code & Fix Model

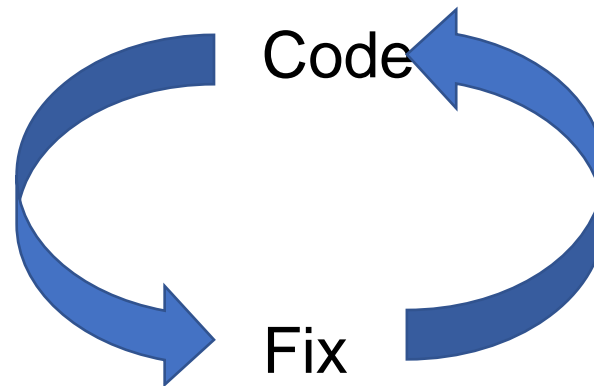
**Code the software & Fix Modify ( similar to trail & error)**

**Generally used for SMALL projects**

**Very short delivery time**

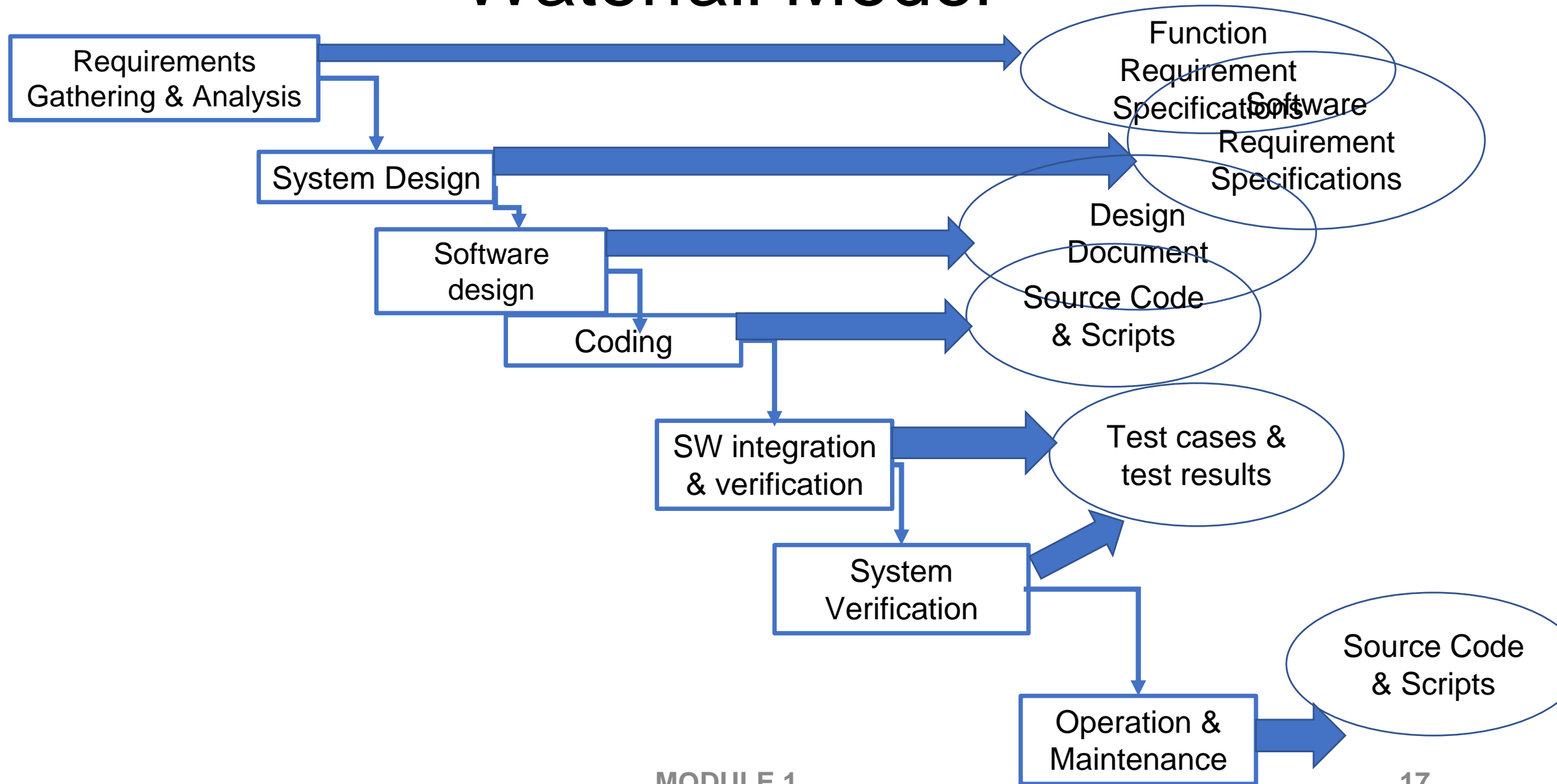
**SE is restricted only to management of program**

**NO way to keep track of requirements, business logic and design**

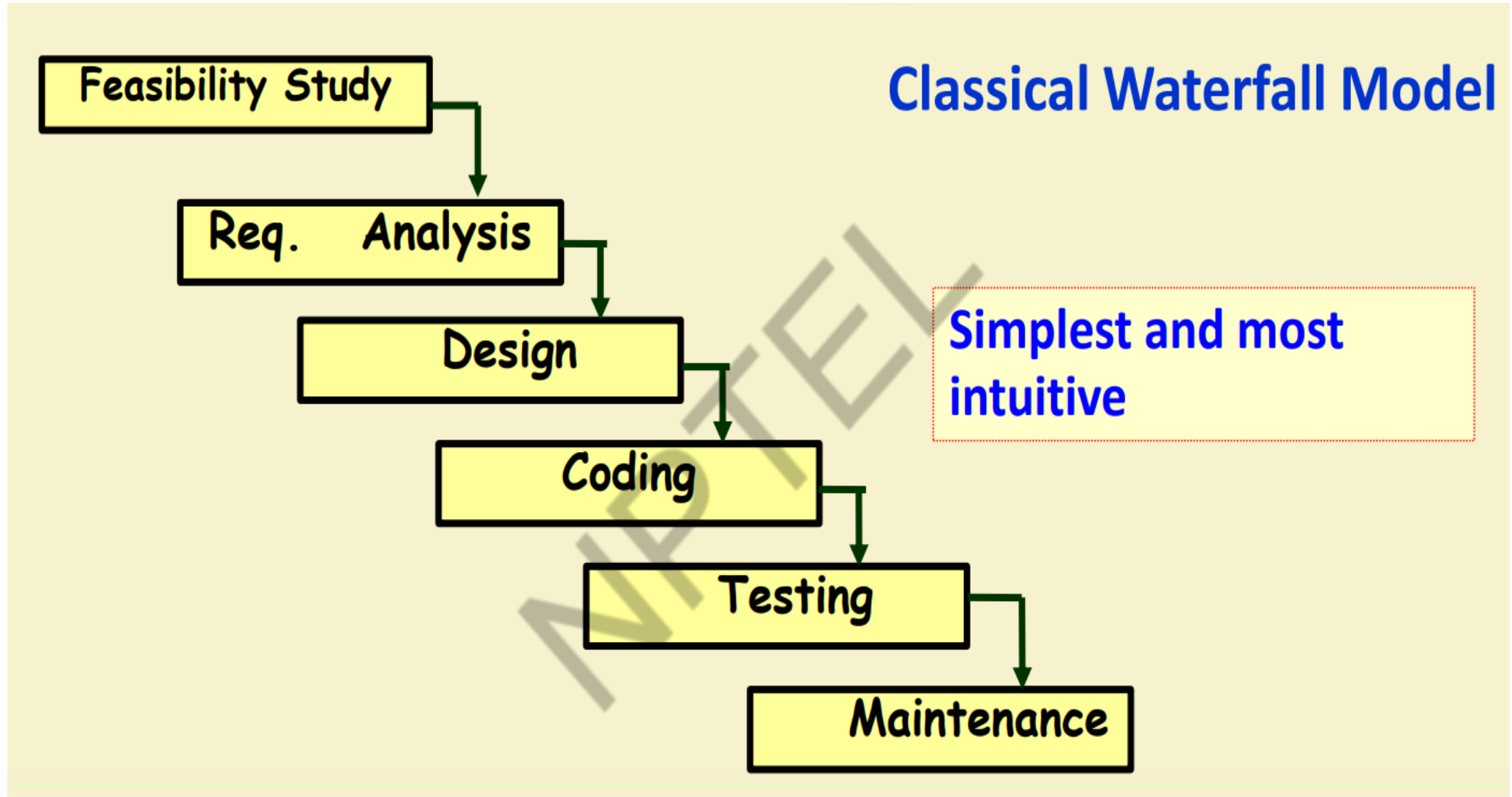




# Waterfall Model



# Waterfall Model



# Types of Maintenance?

- **Corrective maintenance:** –  
Correct errors which were not discovered during the product development phases.
- **Perfective maintenance:** –  
Improve implementation of the system – enhance functionalities of the system.
- **Adaptive maintenance:** –  
Port software to a new environment,  
e.g. to a new computer or to a new operating system.

# When should we use it

1. Requirements are clear.
2. There are no ambiguous requirements
3. It is good to use this model when the technology is well understood.
4. The Project is short and cost is low.
5. Risk is zero and minimum.

# Advantages

- Simple and easy to understand and use.
- Easy to manage
- Works well for smaller and low budget projects where requirements are very well understood.
- Clearly defined stages and well understood.
- Easy to arrange tasks
- Process and results are well documented.

# Disadvantages

- No working software is produced until late during the life cycle.
- High amount of risk and uncertainty.
- Not good model for complex and object oriented project.
- Poor model for long and ongoing projects.
- It is difficult to measure progress within stages.

# Waterfall Model

**Documents are produced at each stage phase**

**Identical and compliance to other engineering process models**

All requirements must be gathered & analyzed at the start  
(not practical)

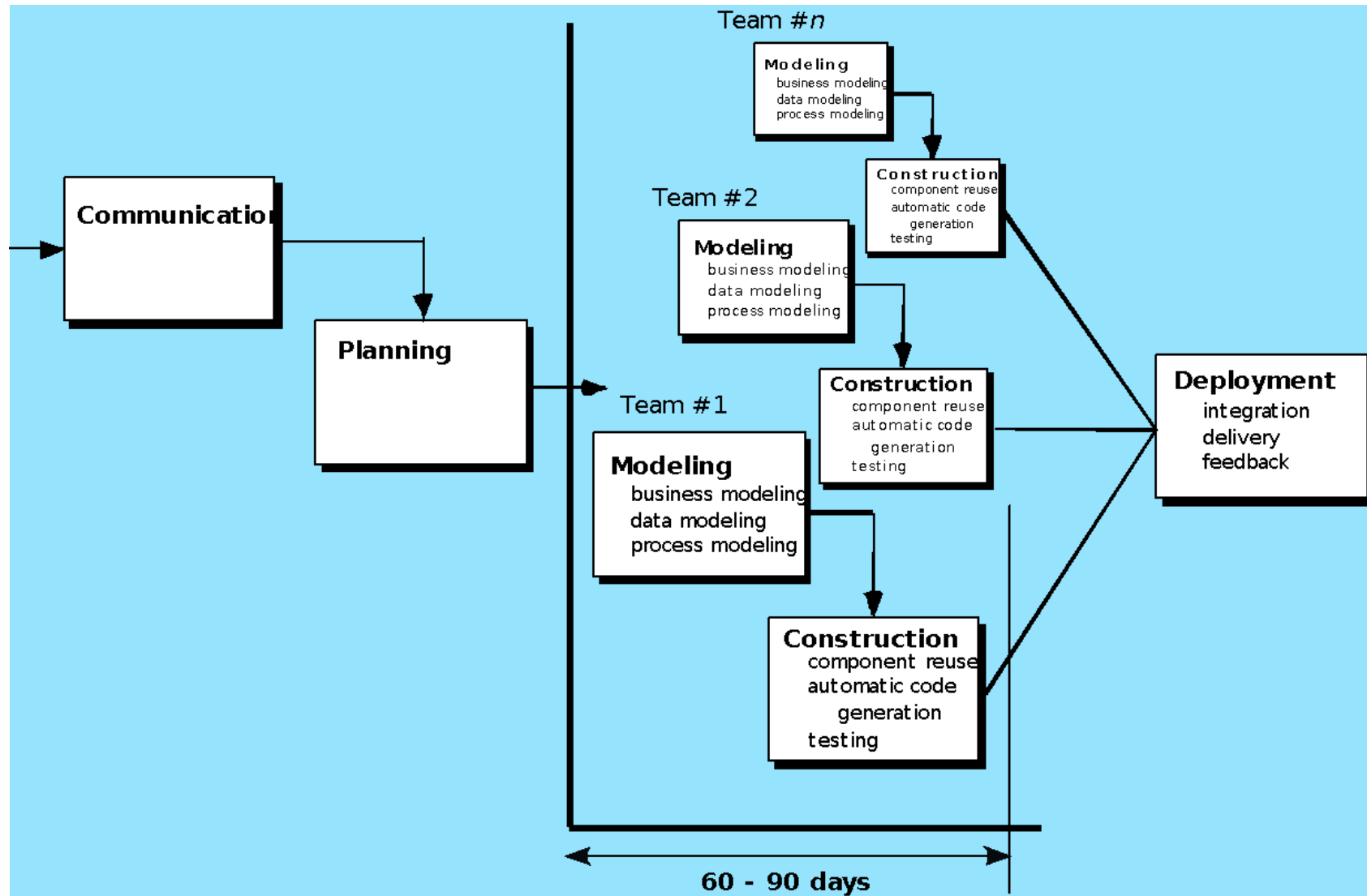
Phase-wise focus

Process restrictions encourages problem postponement  
end up in bad system

Design becomes inflexible

Postponement makes maintenance BIG phase

# The RAD Model



These courseware materials are to be used in conjunction with *Software Engineering: A Practitioner's Approach*, 6/e and are provided with permission by R.S. Pressman & Associates, Inc., copyright © 1996, 2001, 2005



# Rapid Application Development (RAD) Model

**Delivery is available in short duration (within @ 60- 90 days)**

**Multiple teams work in parallel in activities such as modeling , construction of different modules (sub systems)**

**May be inconsistent**

**Integration could be difficult**

# RAD

- Sometimes referred to as the rapid prototyping model.

## Features:

- Decrease the time taken and the cost incurred to develop software systems.
- Facilitate accommodating change requests as early as possible.
  - Before large investments have been made in development and testing.

# RAD

- A way to reduce development time and cost, and yet have flexibility to incorporate changes:
- Make only short term plans and make heavy reuse of existing code.

# Methodology

- Plans are made for one increment at a time.
  - The time planned for each iteration is called a time box.
- Each iteration (increment):
  - Enhances the implemented functionality of the application a little

## During each iteration,

- A quick-and-dirty prototype-style software for some selected functionality is developed.
- The customer evaluates the prototype and gives his feedback.
- The prototype is refined based on the customer feedback.

# How Does RAD Facilitate Faster Development?

- RAD achieves fast creation of working prototypes.
  - Through use of specialized tools.
- These specialized tools usually support the following features:
  - Visual style of development.
  - Use of reusable components.
  - Use of standard APIs (Application Program Interfaces).

# For which Applications is RAD Suitable?

- Customized product developed for one or two customers only
- Performance and reliability are not critical.
- The system can be split into several independent modules.

# Iterative Model

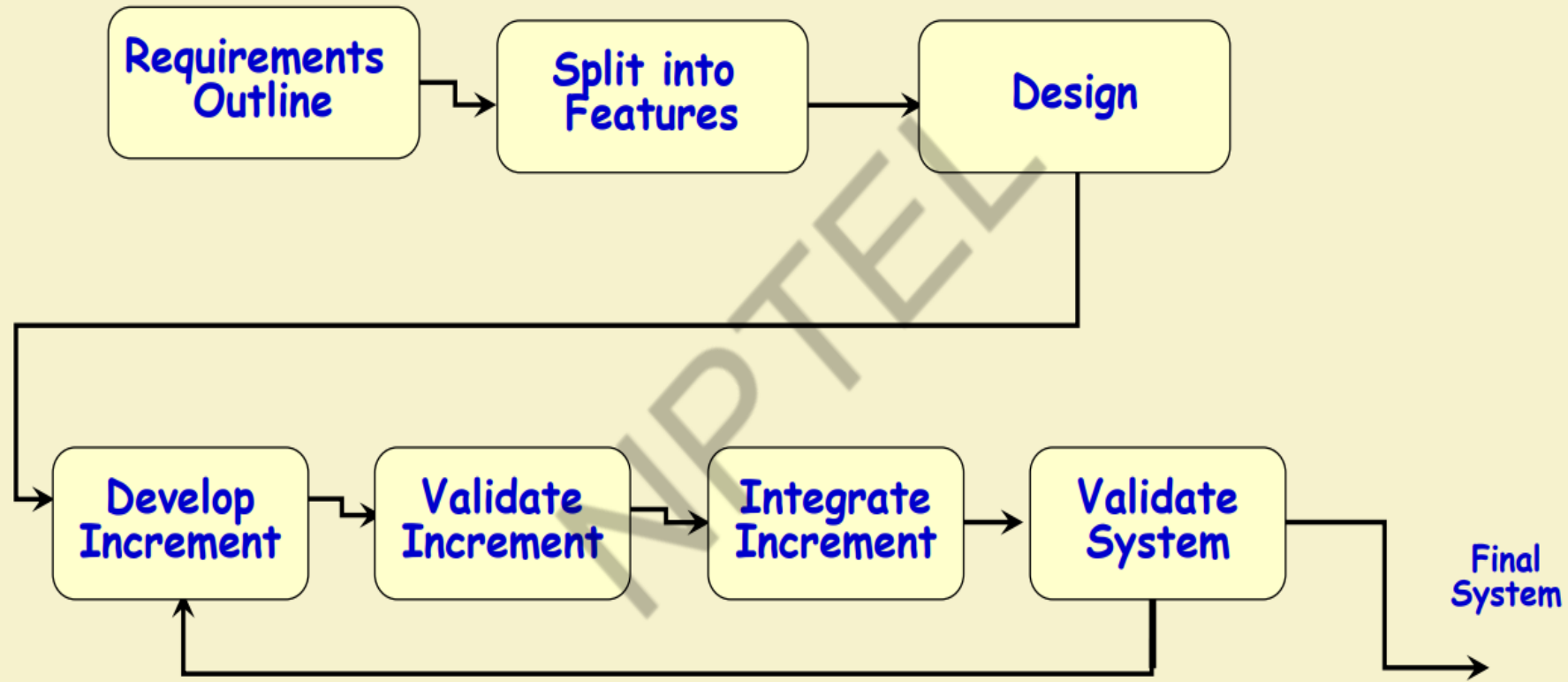
Software development stages are repeated to

**IMPROVE** performance

**Fix bugs (problems)**

Focus is NOT on adding functionalities (as against of Incremental model)

# Incremental Model



Ref: NPTEL course on software engineering, Ranbir mall

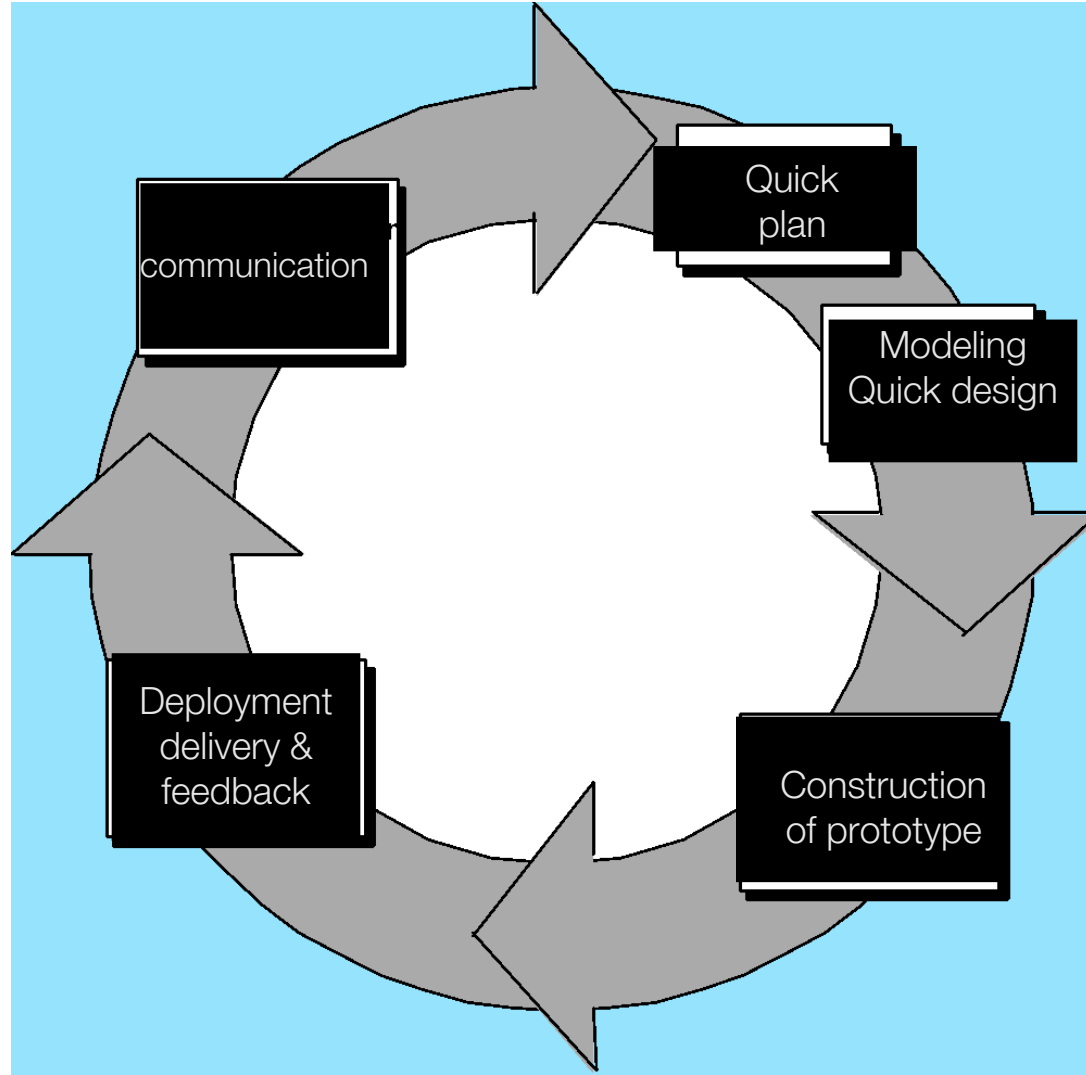


# Incremental model

- Waterfall: single release
  - Iterative: many releases
- 
- First increment: core functionality
  - Successive increments: add/fix functionality
  - Final increment: the complete product

Each iteration: a short mini-project with a separate lifecycle – e.g., waterfall

# Evolutionary Models: Prototyping



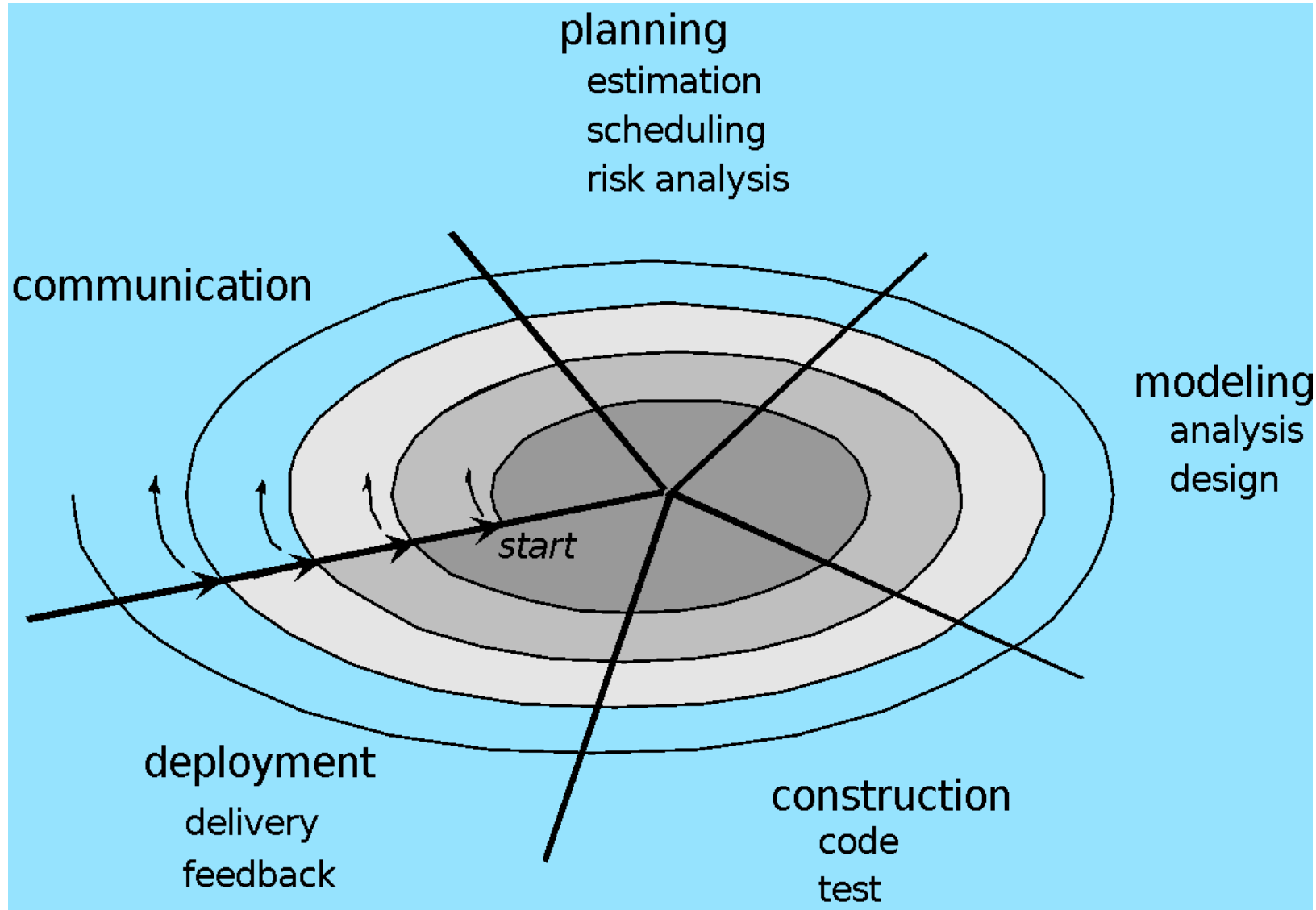
These courseware materials are to be used in conjunction with *Software Engineering: A Practitioner's Approach*, 6/e and are provided with permission by R.S. Pressman & Associates, Inc., copyright © 1996, 2001, 2005

# Evolutionary Prototype Model

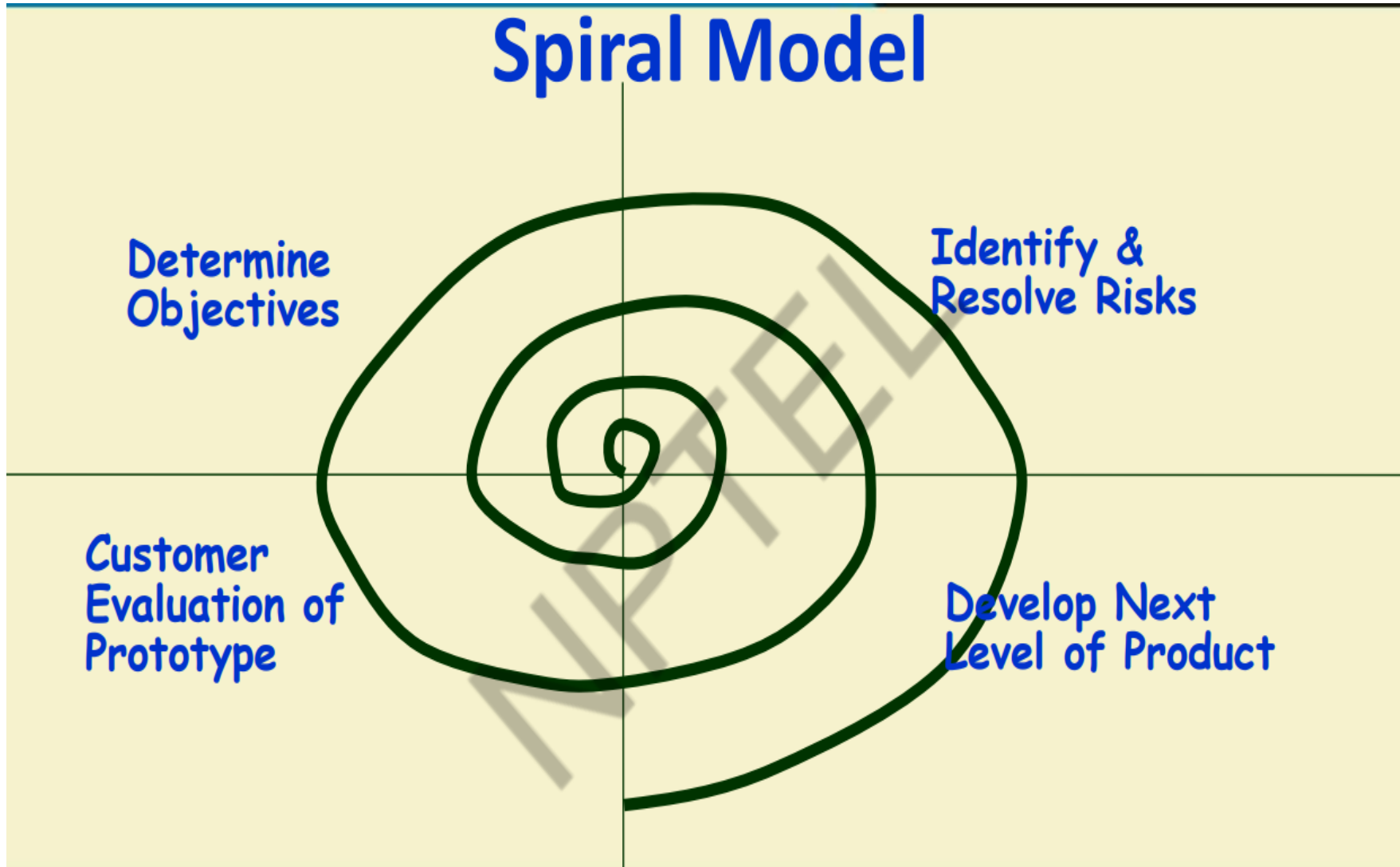
Could be used for understanding  
Requirements  
Functioning  
Testing new technology

The code/ product MAY NOT be used in the final product

# Boehm's Spiral model



# Boehm's Spiral model



# Boehm's Spiral Model

Documents and product are available after each iteration

Improves & adds functionalities after each cycle

Requires ALL types of human resources at regular frequency

# Agile Model

- What is Agile Software Development?

**Agile:** Easily moved, light, nimble, active software processes

- How agility achieved?
  - Fitting the process to the project
  - Avoidance of things that waste time
- To overcome the shortcomings of the waterfall model of development.
  - Proposed in mid-1990s

# Agile Model

- The agile model was primarily designed:
  - To help projects to adapt to change requests
- In the agile model:
  - The requirements are decomposed into many small incremental parts that can be developed over one to four weeks each.



# Agile

- Effective (rapid and adaptive) **response to change** (team members, new technology, requirements)
- Effective **communication** in structure and attitudes among all team members, technological and business people, software engineers and managers
- Involving **the customer into the team**
- Working as a **team**
- Planning in an uncertain world has its limits and plan must be **flexible**
- Organizing a **team** so that it **is in control of the work performed**
- Emphasize an **incremental delivery** strategy as opposed to intermediate products that gets working software to the customer as rapidly as feasible.
- Rapid**, incremental delivery of software
- The development guidelines stress **delivery over analysis and design** although these activates are not discouraged, and **active and continuous communication** between developers and customers

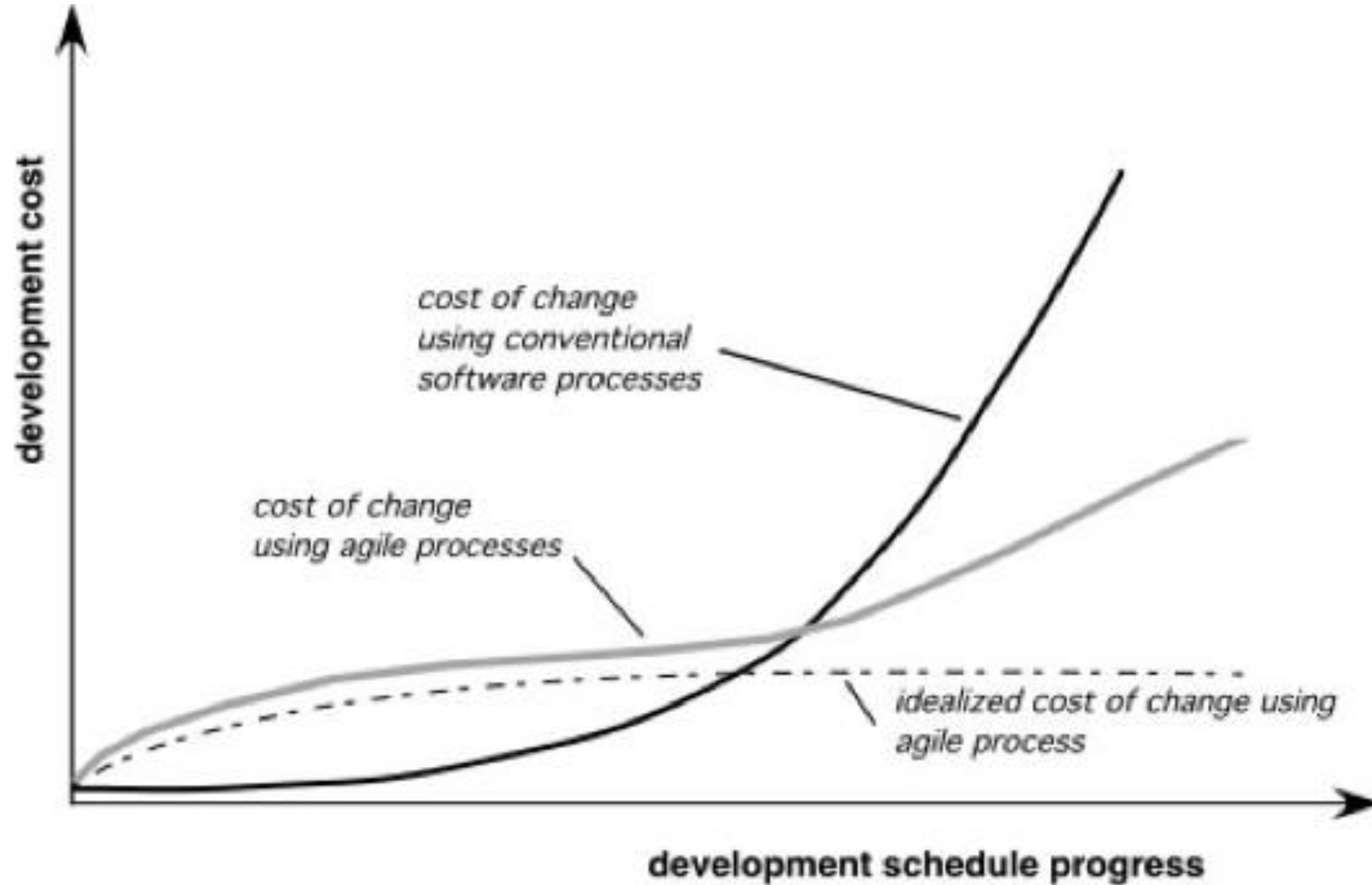
# Benefits of Agility

Relatively easy to accommodate a change when a team is gathering requirements early in a project.

The costs of doing this work are minimal.

A well-designed agile process may “flatten” the cost of change curve by coupling incremental delivery with agile practices such as continuous unit testing and pair programming.

# Cost of Change in Agile Approach



# Agile Process

- A **customer descriptions** of what is required (scenarios). Some assumptions:
- Recognizes that plans are short-lived
  - some requirements will persist
  - some will change
  - Customer priorities will change
- Develops software iteratively with a heavy **emphasis on construction** activities
  - Design and construction are interleaved, hard to say how much design is necessary before construction
  - Design models are proven as they are created
  - Analysis, design, construction and testing are not predictable
- Adapt as changes occur due to unpredictability
- Delivers multiple 'software increments', deliver an operational prototype or portion of an OS to collect customer feedback for adaption.

# Principles of Agility

1. Highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Allow changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale
4. Business people and developers must work together daily throughout the project
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation

# Principles of Agility

7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely
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. The best architectures, requirements, and designs emerge from self-organizing teams
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly

# Human factors in Agile Development Process

Development would be people focused than process

- **Competence** (talent, skills, knowledge)
- **Common focus** (deliver a working software increment )
- **Collaboration** (peers and stakeholders)
- **Decision-making ability** (freedom to control its own destiny)
- **Fuzzy problem-solving ability** (ambiguity and constant changes, today problem may not be tomorrow's problem)
- **Mutual trust** and respect
- **Self-organization** (themselves for the work done, process for its local environment, the work schedule)

# Agile Methodologies

2016 Q1  
2017 Q2

- XP
- Scrum
- Unified process
- Crystal
- DSDM
- Lean



# Agile Approach Extreme Programming

- The most widely used agile process
  - **XP Planning**
  - **XP Design**
  - **XP Coding**
  - **XP Testing**

# Extreme Programming (XP)

- The most widely used agile process

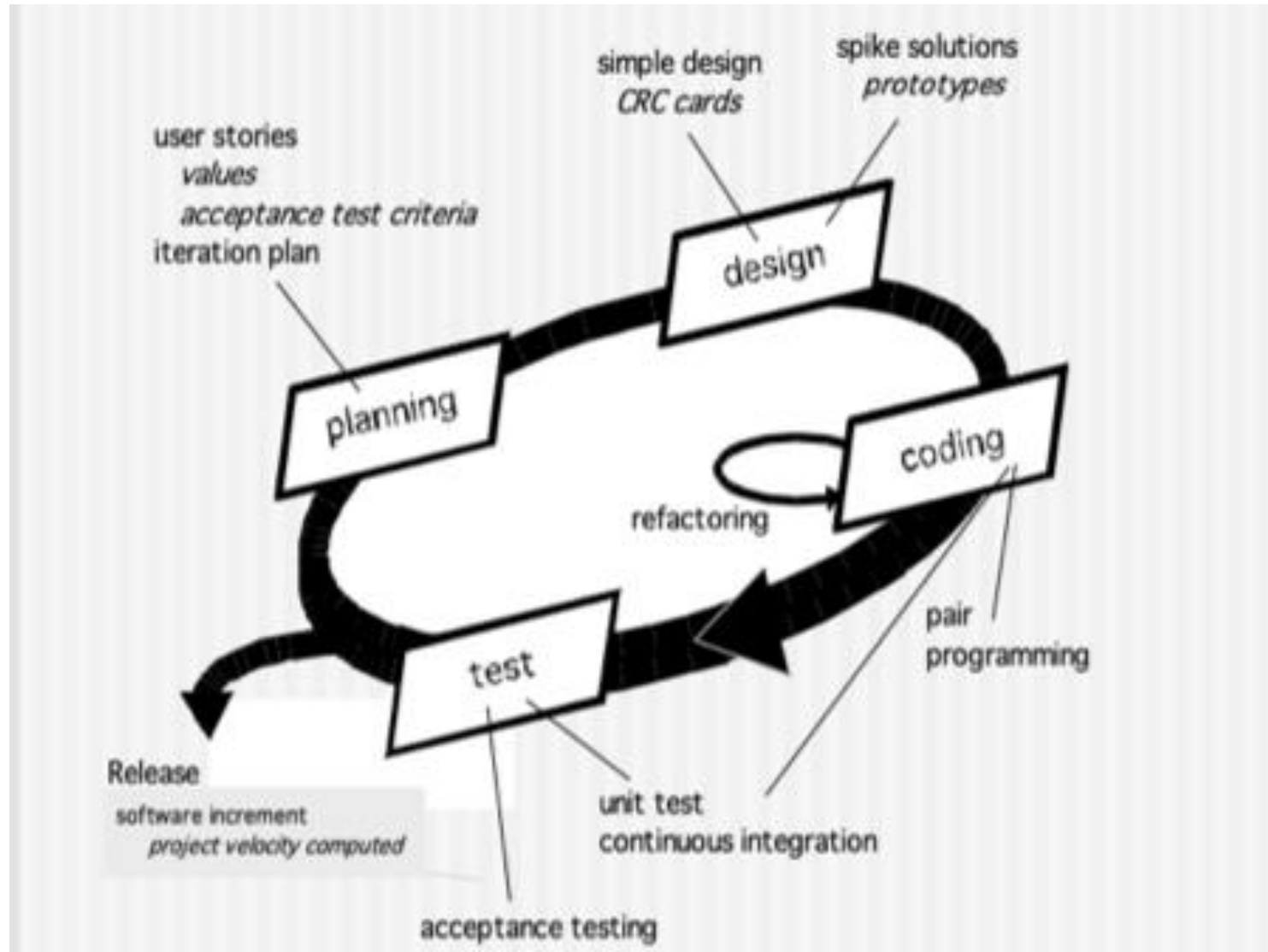
- XP Planning**

- Begins with the listening, leads to creation of “**user stories**” that describes required output, features, and functionality. Customer assigns a value(i.e., a priority) to each story.
  - Agile team assesses each story and assigns a cost (development weeks. If more than 3 weeks, customer asked to split into smaller stories)
  - Working together, stories are grouped for a deliverable increment next release.
- A **commitment** (stories to be included, delivery date and other project matters) is made. Three ways:
  - 1.Either all stories will be implemented in a few weeks,
  - 2.High priority stories first, or
  - 3.the riskiest stories will be implemented first.
- After the first increment “**project velocity**” (number of stories implemented during the first release) is used to help define subsequent delivery dates for other increments. Customers can add stories, delete existing stories, change values of an existing story, split stories as development work proceeds.

# Extreme Programming (XP)

- **XP Design** ( occurs both before and after coding as refactoring is encouraged)!
  - Follows the Keep It Simple(KIS) principle)Nothing more nothing less than the story.
  - Encourage the use of **CRC (class-responsibility-collaborator)** cards in an object-oriented context. The only design work product of XP. They identify and organize the classes that are relevant to the current software increment
  - For difficult design problems, suggests the creation of “**spike solutions**”—a design prototype for that portion is implemented and evaluated.
  - Encourages “**refactoring**”—an iterative refinement of the internal program design. Does not alter the external behavior yet improve the internal structure. Minimize chances of bugs. More efficient, easy to read
- **XP Coding**
  - Recommends the construction of a unit test for a story before coding commences. So implementer can focus on what must be implemented to pass the test
  - Encourages “**pair programming**”. Two people work together at one workstation. Real time problem solving, real time review for quality assurance. Take slightly different roles
- **XP Testing**
  - All unit tests are executed daily and ideally should be automated. Regression tests are conducted to test current and previous components
  - “Acceptance tests” are defined by the customer and executed to assess customer visible functionality

# Extreme Programming (XP)



# Merits & Limitations of XP

- **Requirements volatility:** customer is an active member of XP team, changes to requirements are requested informally and frequently
- **Conflicting customer needs:** different customers' needs need to be assimilated. Different vision or beyond their authority
- **Requirements are expressed informally:** Use stories and acceptance tests are the only explicit manifestation of requirements. Formal models may avoid inconsistencies and errors before the system is built. Proponents said changing nature makes such models obsolete as soon as they are developed
- **Lack of formal design:** XP deemphasizes the need for architectural design. Complex systems need overall structure to exhibit quality and maintainability. Proponents said incremental nature limits complexity as simplicity is a core value

# Crystal

A resource limited, cooperative game of invention and communication, with a primary goal of delivering useful, working software and a secondary goal of setting up for the next game.

A set of methodologies, each with core elements that are common to all, and roles, process patterns, work products, and practice that are unique to each

## Distinguishing features:

- A collection of process models that allow “maneuverability” based on problem characteristics.
- Face-to-face communication is emphasized.
- Suggests the use of “reflection workshops” to review the work habits of the team.

# Adaptive Software Development (ASD)

Focusing on human collaboration and team self-organization as a technique to build complex software and system.

## Distinguishing features

- **Mission-driven** planning
- **Component-based** focus
- Uses “**time-boxing**” focus of Project Management Concepts
- Explicit **consideration of risks**
- Emphasizes **collaboration** for requirements gathering
- Emphasizes “**learning**” throughout the process

# Three Phases of ASD

1. **Speculation:** project is initiated and adaptive cycle planning is conducted.
  - Adaptive cycle planning uses project initiation information- the customer's mission statement, project constraints (e.g. delivery date), and basic requirements to define the set of release cycles (increments) that will be required for the project.
  - Based on the information obtained at the completion of the first cycle, the plan is reviewed and adjusted so that planned work better fits the reality.



# Three Phases of ASD

**2. Collaborations** are used to multiply their talent and creative output beyond absolute number ( $1+1>2$ ).

It encompasses communication and teamwork, but it also emphasizes individualism, because individual creativity plays an important role in collaborative thinking

It is a matter of trust.

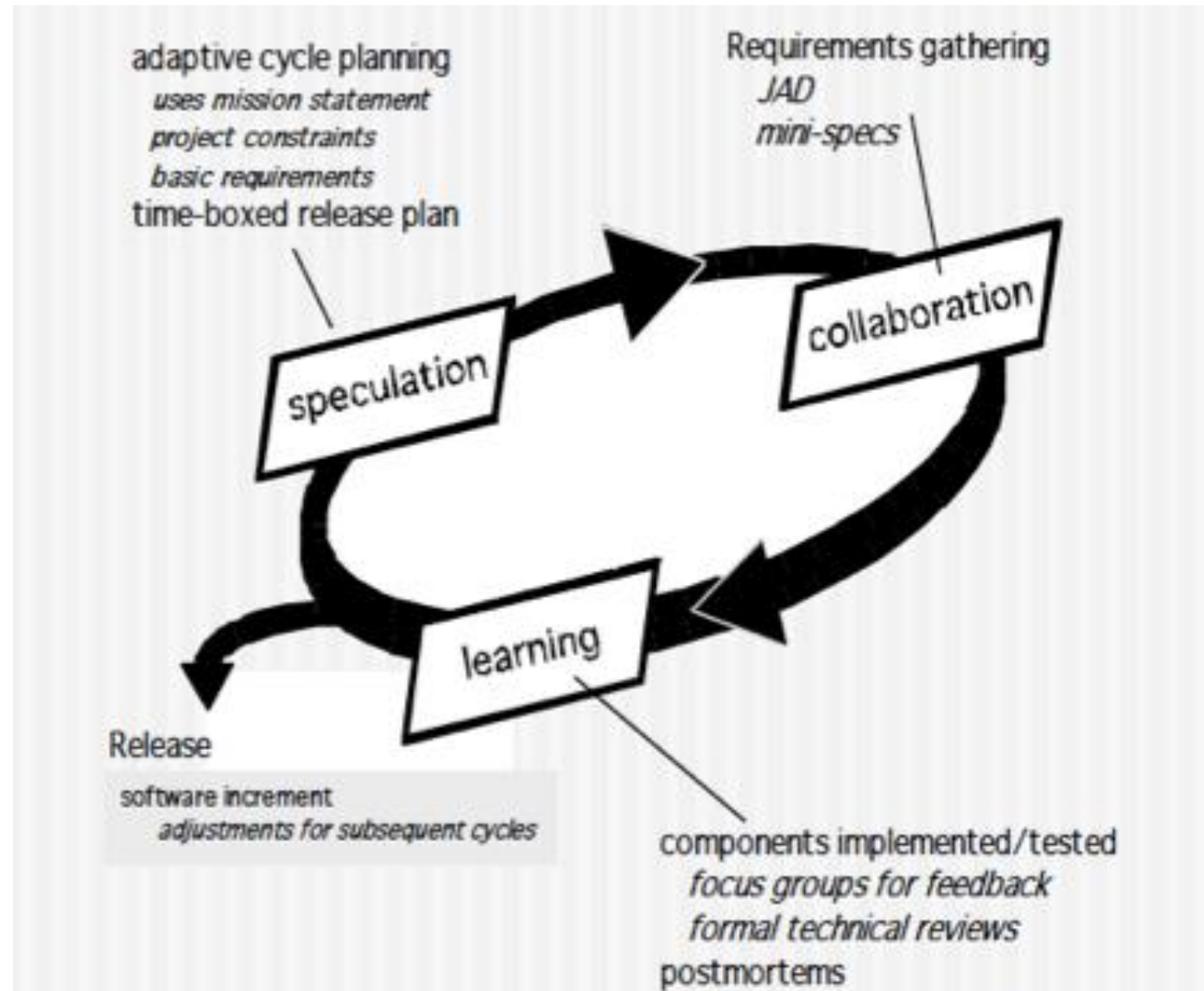
- criticize without animosity
- assist without resentments
- work as hard as or harder than they do
- have the skill set to contribute to the work at hand
- communicate problems or concerns in a way that leads to effective action.

# Three Phases of ASD

**3. Learning:** As members of ASD team begin to develop the components, the emphasis is on “learning”.

- software developers often overestimate their own understanding of the technology, the process, and the project and that learning will help them to improve their level of real understanding.
- Three ways: focus groups, technical reviews and project post-mortems

# Three Phases of ASD



# Dynamic Systems Development Method (DSDM)

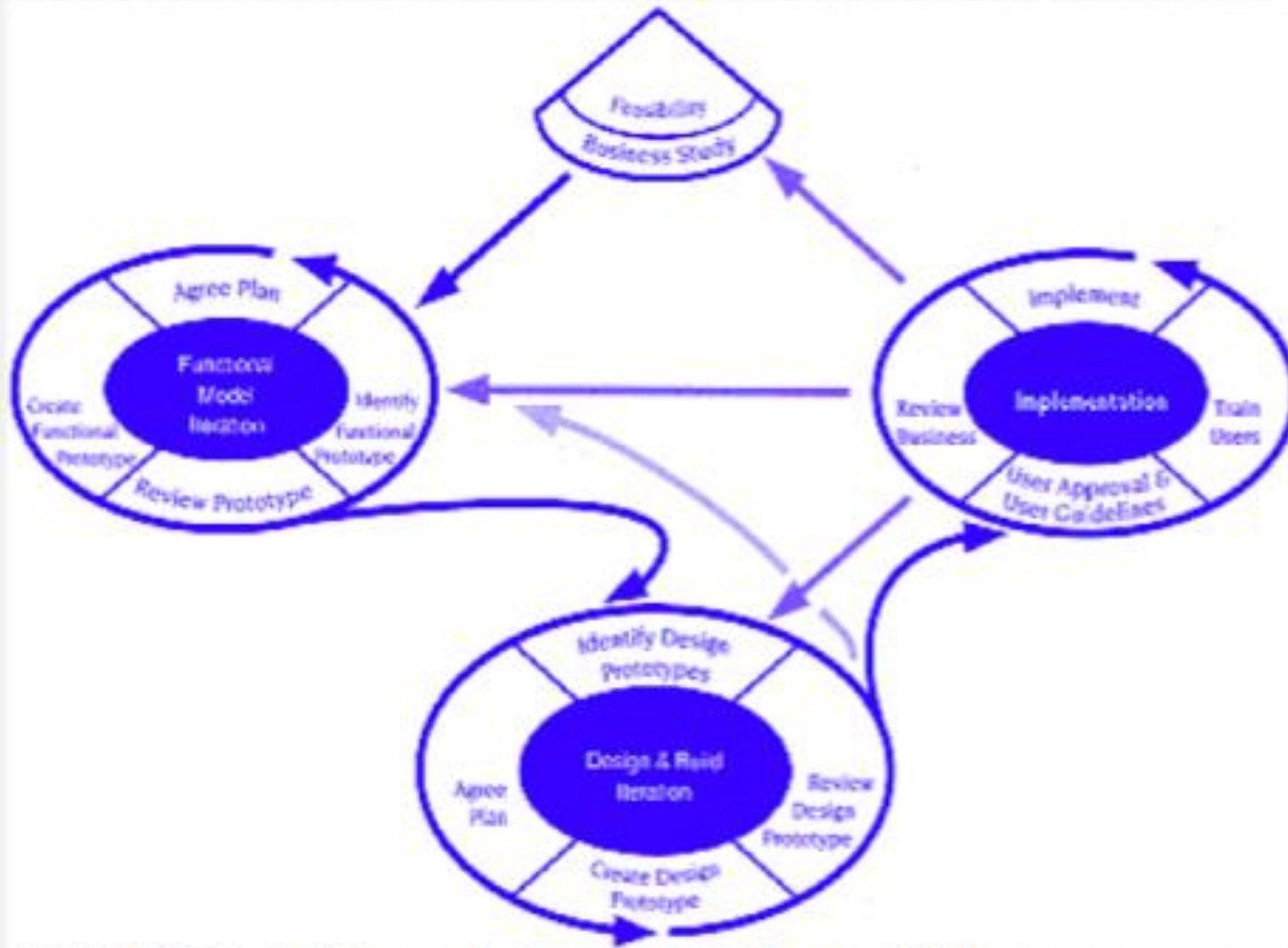
Provides a framework for building and maintaining systems which meet tight time constraints through the use of incremental prototyping in a controlled project environment

- DSDM—distinguishing features
- Similar in most respects to XP and/or ASD

- **Guiding principles**

1. Active user involvement is imperative.
2. DSDM teams must be empowered to make decisions
3. The focus is on frequent delivery of products
4. Fitness for business purpose is the essential criterion for acceptance of deliverables
5. Iterative and incremental development is necessary to converge on an accurate business solution
6. All changes during development are reversible
7. Requirements are baselined at a high level
8. Testing is integrated throughout the life-cycle

# DSDM Life Cycle



**DSDM Life Cycle (with permission of the DSDM consortium)**

These slides are designed to accompany Software Engineering & Quality Assurance Approach 71

# Scrum

## Features

- Development work is partitioned into “**packets**”
- Testing and documentation are on-going** as the product is constructed
- Work units occurs in “**sprints**” and is derived from a “**backlog**” of existing changing prioritized requirements
- Changes are not introduced in sprints (short term but stable) but in backlog
- Meetings are very short** (15 minutes daily) and sometimes conducted without chairs ( what did you do since last meeting? What obstacles are you encountering? What do you plan to accomplish by next meeting?)
- “**demos**” are delivered to the customer with the time-box allocated. May not contain all functionalities. So customers can evaluate and give feedbacks.

# Assignment

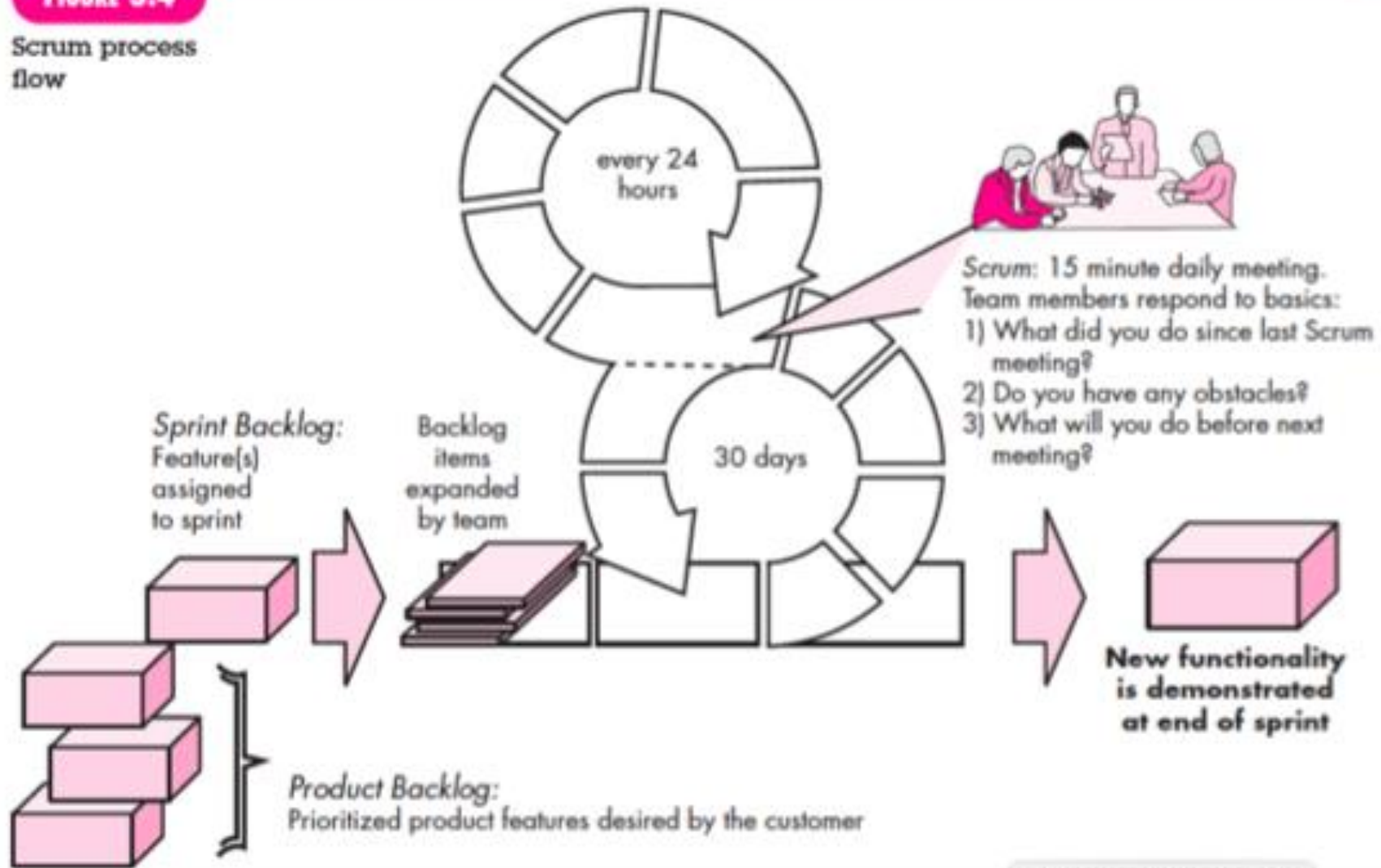
**Compare various Software Development Life Cycle models**

Linear, Iterative, Spiral, Prototype, Agile

# Scrum

**FIGURE 3.4**

Scrum process flow



Adobe Acrobat Reader DC



# Feature Driven Development (FDD)

Features:

Emphasis is on defining “**features**” which can be organized hierarchically  
a **feature** “is a client-valued function that can be implemented in two weeks or less.”

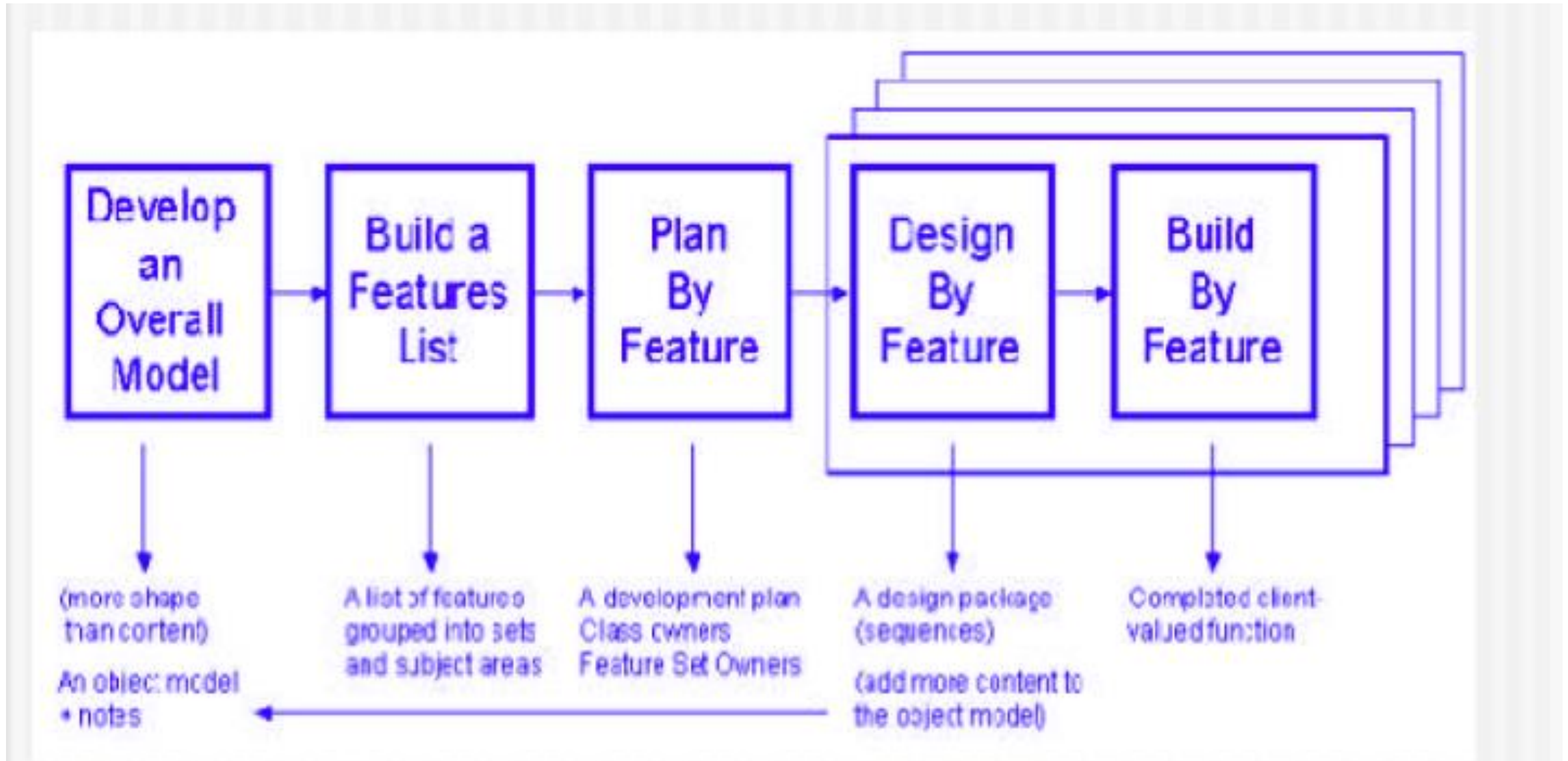
Uses a **feature template**

- <action> the <result> <by | for | of | to> a(n) <object>!
- E.g. Add the product to shopping cart.
- Display the technical-specifications of the product.
- Store the shipping-information for the customer.

A **features list** is created and “**plan by feature**” is conducted

Design and construction merge in FDD

# FDD



# Software Development Process focus

- Use Appropriate
  - Tools
  - Methods
  - Process models

**To Develop a quality product**

# Steps involved in Software Development

1. Identification of Problem
2. Problem Definition
3. Exploring approach(s) to solve the problem
4. Penning down/ finalizing the approach; listing the steps
5. Development
6. Testing
7. Deployment
8. Maintenance

These steps are iterative and some of them run parallel.

# Software Process

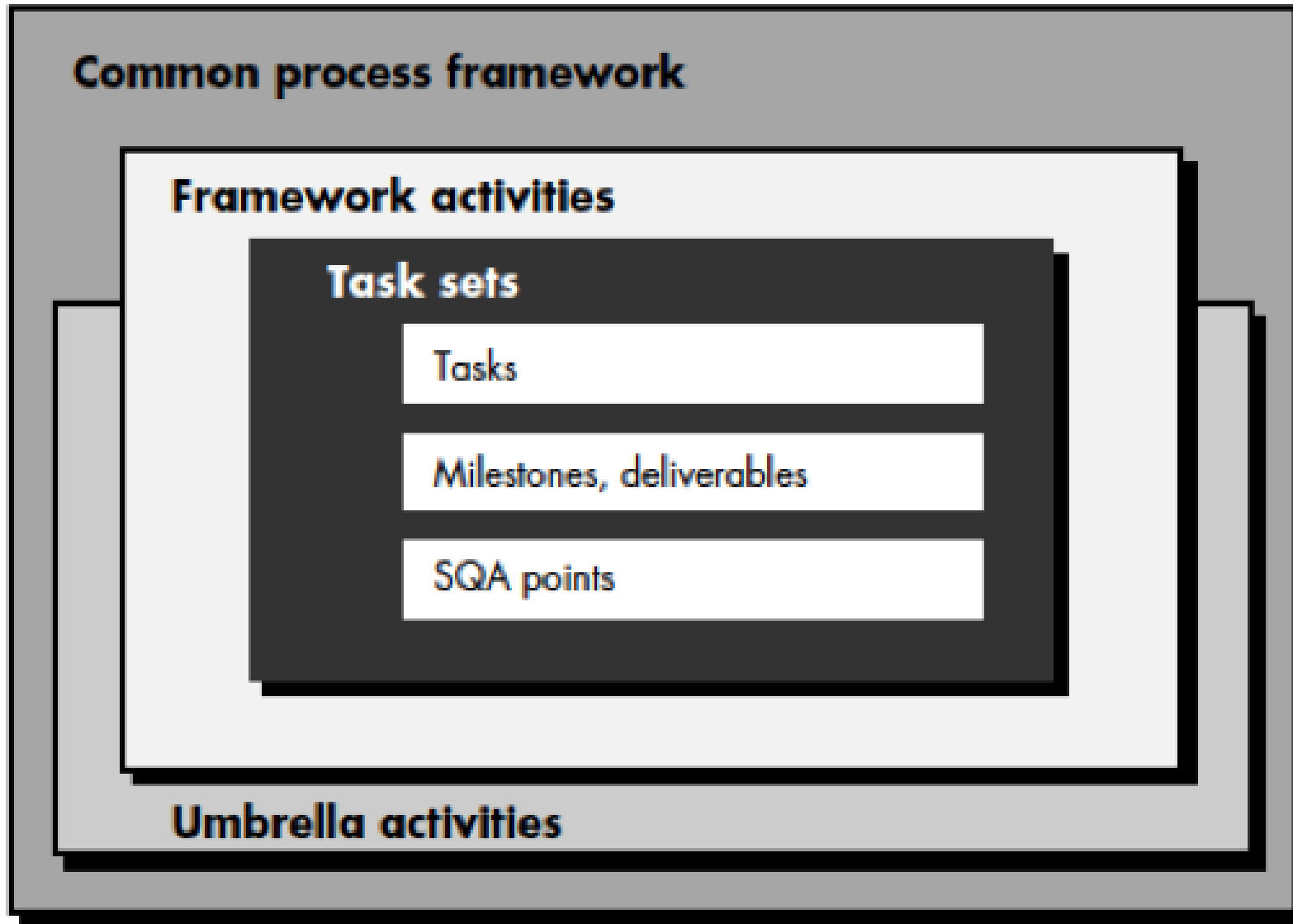
## Characterized by:

- Common process framework established by defining a small number of framework activities that are applicable to all software projects, regardless of their size or complexity

## Consists:

- Framework Activities which include:
- Task sets, represented by Tasks, Milestones & Deliverables , SQA points
- Umbrella Activities

# Process Framework activities



# Process Framework activities

- Work tasks
- Work products
- Milestones & Deliverables
- Quality Assurance Check points

The following *generic process framework* is applicable to the vast majority of S/W projects.

- **Communication:** involves heavy communication with the customer (and other stakeholders) and encompasses requirements gathering.
- **Planning:** Describes the technical tasks to be conducted, the risks that are likely, resources that will be required, the work products to be produced and a work schedule.
- **Modeling:** encompasses the creation of models that allow the developer and customer to better understand S/W req. and the design that will achieve those req.
- **Construction:** combines code generation and the testing required uncovering errors in the code.
- **Deployment:** deliver the product to the customer who evaluates the delivered product and provides feedback.

# Process Umbrella activities

Typical Umbrella activities include:

- Software project tracking and control
- Formal technical reviews
- Software quality assurance
- Software configuration management
- Document preparation and production
- Reusability management
- Measurement
- Risk management

**Umbrella activities are applied throughout the software process**



# Metrics, Measures & Indicators

**Metrics** represent the different methods employed to understand change over time across a number of dimensions or criteria.

It is often used as a catch all terms to describe the method used to measure something, the resulting values obtained from measuring, as well as a calculated or combined set of measures.

**Measures** is a number or a quantity that records a directly observable value or performance.

**Indicators are** qualitative or quantitative factors or variables that provide a simple and reliable meaning to express achievement, the attainment of a goal, or the results stemming from a specific change

# Process Maturity

Software Engineering Institute (SEI) has developed a comprehensive model predicated on a set of software engineering capabilities that should be present for an organization when the organization reaches different levels of process maturity.

**Five** levels usually it is termed as Capability Maturity Model (CMM) Levels.

## Software Engineering:

- Aims to produce a Quality product
- Focuses on Process
- Systematic process which leads to Quality product

**Assessment of Process and Product** is represented and interpreted using

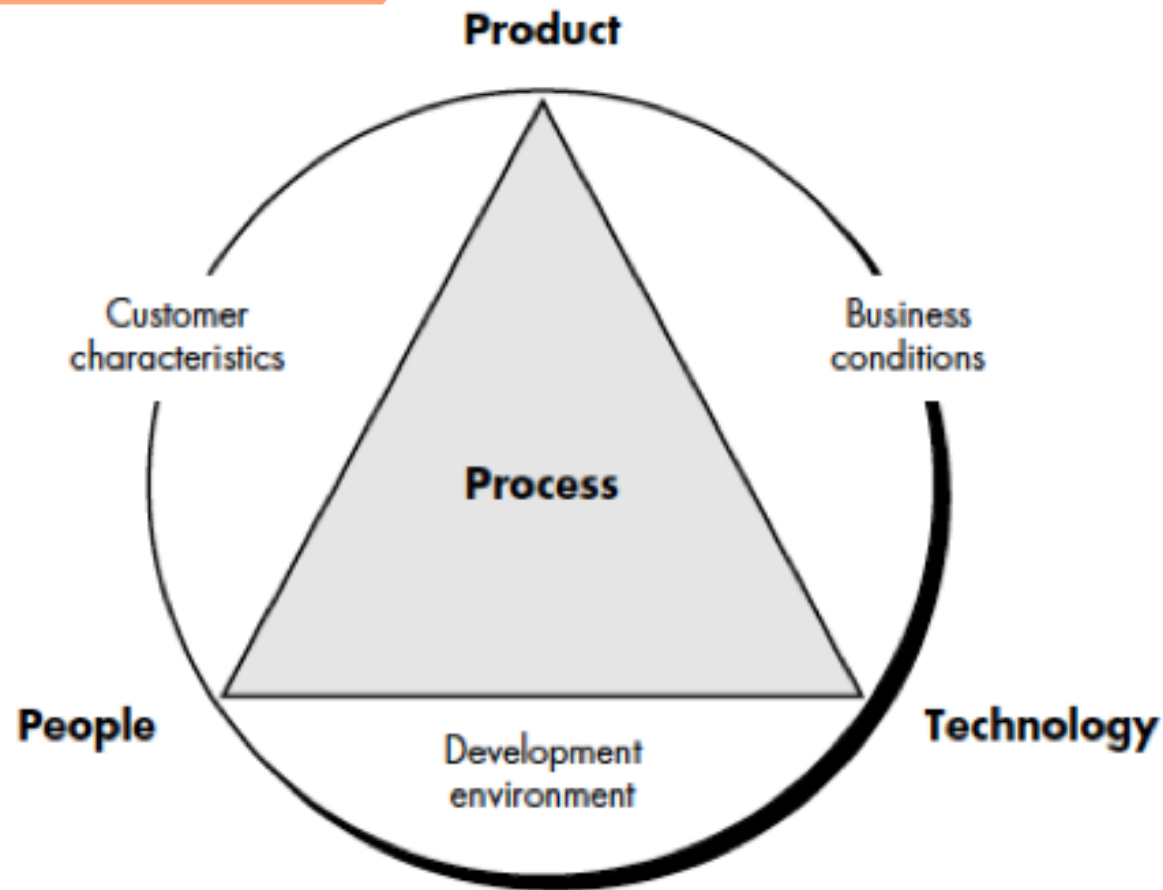
- Process Metrics
- Product Metrics

# Process Metrics

Pertain to **Process Quality**

Used to **measure** the **efficiency** and **effectiveness** of **various processes**

Used for **strategic purpose**



# Software Process Metrics

Software process metrics can provide significant benefit as an organization works to improve its overall level of process maturity.

Software metrics etiquette are:

Use common sense and organizational sensitivity when interpreting metrics data.

Provide regular feedback to the individuals and teams who collect measures and metrics.

Don't use metrics to appraise individuals.

Work with practitioners and teams to set clear goals and metrics that will be used to achieve them.

Never use metrics to threaten individuals or teams.

Metrics data that indicate a problem area should not be considered "negative." These data are merely an indicator for process improvement.

Don't obsess on a single metric to the exclusion of other important metrics. These steps are iterative and some of them run parallel.

# Software Process Metrics

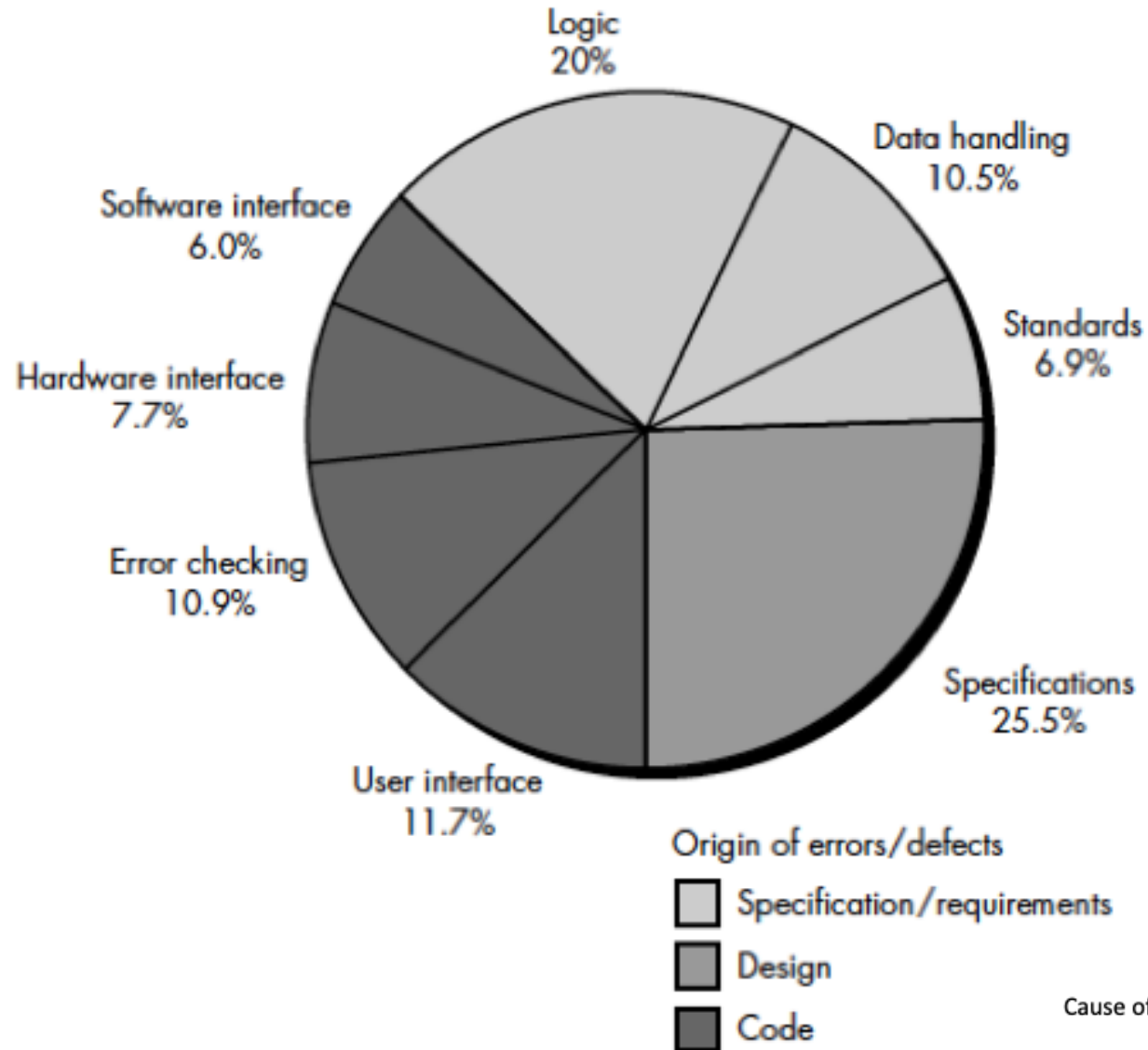
## Statistical software process improvement (SSPI):

- the collection and use of process metrics, the derivation of simple indicators gives way to a more rigorous approach.
- SSPI uses software failure analysis to collect information about all errors and defects encountered as an application, system, or product is developed and used.

## Failure analysis works in the following manner:

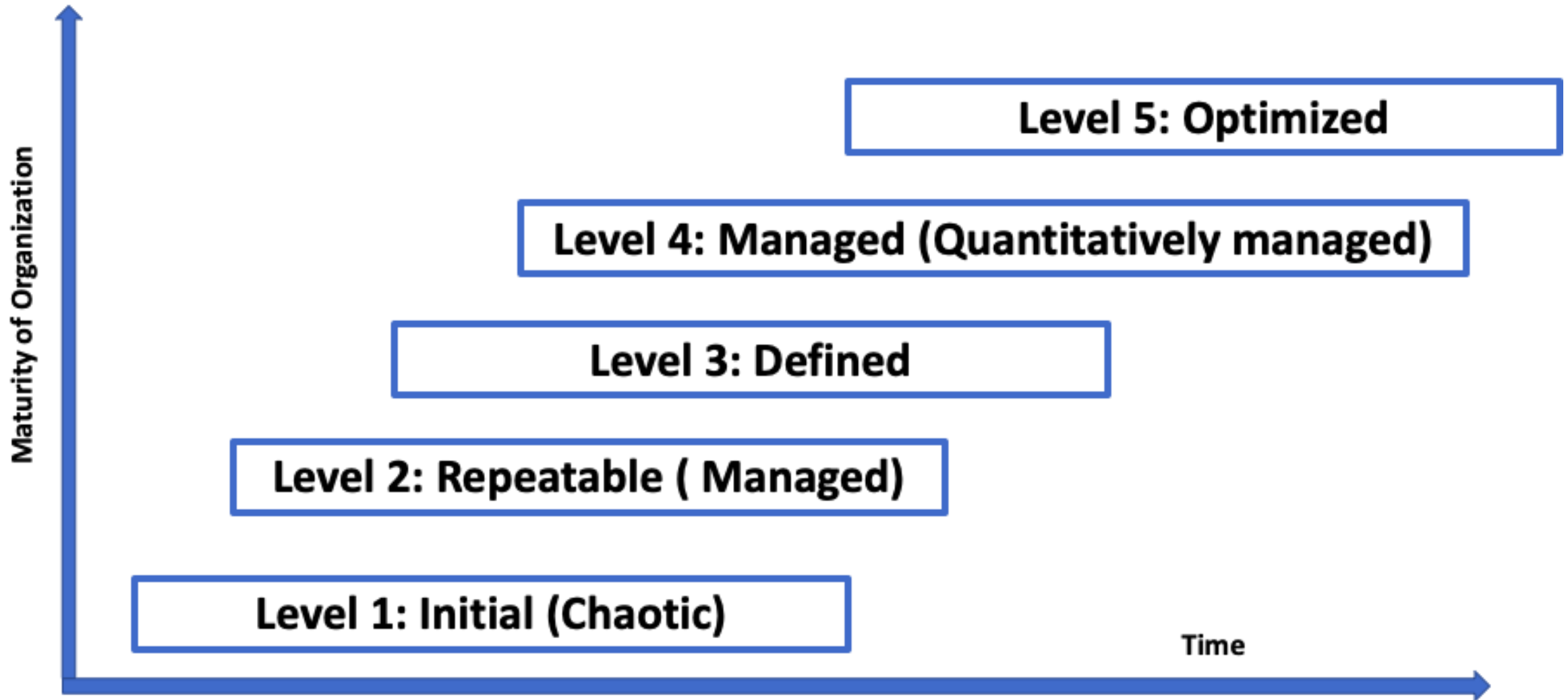
1. All errors and defects are categorized by origin (e.g., flaw in specification, flaw in logic, non-conformance to standards).
2. The cost to correct each error and defect is recorded.
3. The number of errors and defects in each category is counted and ranked in descending order.
4. The overall cost of errors and defects in each category is computed.
5. Resultant data are analyzed to uncover the categories that result in highest cost to the organization.
6. Plans are developed to modify the process with the intent of eliminating

# Failure Analysis



Cause of Defects & Their origin adapted from GRA94

# CMM Levels





# CMM Levels

- **Level 1: Initial.** The software process is characterized as ad hoc and occasionally even chaotic. Few processes are defined, and success depends on individual effort.
- **Level 2: Repeatable (Managed).** Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.
- **Level 3: Defined.** The software process for both management and engineering activities is documented, standardized, and integrated into an organization-wide software process. All projects use a documented and approved version of the organization's process for developing and supporting software. This level includes all characteristics defined for level 2.
- **Level 4: Managed (Quantitatively managed).** Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled using detailed measures. This level includes all characteristics defined for level 3.
- **Level 5: Optimizing.** Continuous process improvement is enabled by quantitative feedback from the process and from testing innovative ideas and technologies. This level includes all characteristics defined for level 4.

# Key Process Areas (KPA)

- The SEI has associated key process areas (KPAs) with each of the maturity levels which describe those software engineering functions (e.g., software project planning, requirements management) that must be present to satisfy good practice at a particular level.

- Characteristics Of KPAs**

- Goals**—the overall objectives that the KPA must achieve.

- Commitments**—requirements (imposed on the organization) that must be met to achieve the goals or provide proof of intent to comply with the goals.

- **Abilities**—those things that must be in place (organizationally and technically) to enable the organization to meet the commitments.

- Activities**—the specific tasks required to achieve the KPA function.

- Methods for monitoring implementation**—the manner in which the activities are monitored as they are put into place.

- Methods for verifying implementation**—the manner in which proper practice for the KPA can be verified.

# KPAs of CMM Level 2

- **Process maturity level 2 (Repeatable /managed)**

Goals, Commitments, Abilities, Activities, Methods for monitoring implementation, Methods for verifying implementation

- Software configuration management
- Software quality assurance
- Software subcontract management
- Software project tracking and oversight
- Software project planning
- Requirements management

# KPAs of CMM Level 3

- **Process maturity level 3 (Defined)**

Goals, Commitments, Abilities, Activities, Methods for monitoring implementation, Methods for verifying implementation achieved through:

- Peer reviews
- Intergroup coordination
- Software product engineering
- Integrated software management
- Training program
- Organization process definition
- Organization process focus

# KPAS of CMM Level 4 & 5

- **Process maturity level 4 Managed (Quantitatively managed)**

Goals, Commitments, Abilities, Activities, Methods for monitoring implementation, Methods for verifying implementation

- Software quality management
- Quantitative process management

- **Process maturity level 5 (Optimized)**

- Process change management
- Technology change management
- Defect prevention

# 1.3 Software Planning & Estimation

Accurate estimation of project size is central to satisfactory estimation of all other project parameters such as effort, completion time, and total project cost.

- Product Metrics estimations : LOC, FP, COCOMO .

**LOC : This metric measures the size of a project by counting the number of source instructions in the developed program.**

**FP(Function point metric): FP has several advantages over LOC.**

**FP can be easily computed from the problem specification itself.**

**Using LOC, size can be accurately determined only after code can be fully written.**

# Software **Product** metrics

- Software project measures are tactical; project metrics and the indicators derived from them are used by a project manager and a software team to adapt project workflow and technical activities.
- The first application of project metrics on most software projects occurs during estimation.
- Metrics collected from past projects are used as a basis from which effort and time estimates are made for current software work.
- As a project proceeds, measures of effort and calendar time expended are compared to original estimates (and the project schedule). The project manager uses these data to monitor and control progress.

## **Used to :**

1. **minimize the development schedule** by making the adjustments necessary to avoid delays and mitigate potential problems and risks.
2. **assess product quality** on an ongoing basis and, when necessary, modify the technical approach to improve quality.

# Software **Measurement**

- **Direct measures** of the **software engineering process** include **cost and effort** applied.
- **Direct measures** of the **product** include **lines of code (LOC) produced, execution speed, memory size, and defects** reported over some **set period of time**.
- **Indirect measures** of the **product** include **functionality, quality, complexity, efficiency, reliability, maintainability**



# Types of Software **product** Metrics

## 1. **Size-Oriented Metrics**

- metrics derived by normalizing quality and/or productivity measures by considering the size of the software that has been produced.

- a set of simple size-oriented metrics can be developed for each project:

- Errors per KLOC (thousand lines of code).
- Defects per KLOC.
- \$ per LOC.
- Page of documentation per KLOC.

In addition, other interesting metrics can be computed:

- Errors per person-month.
- LOC per person-month.
- \$ per page of documentation.


# Product Metrics

## 2. Function-oriented Metrics

- software metrics use a **measure** of the **functionality** delivered by the application
- Function points **derived** from **direct measures** of the **information domain**.
  1. **Number of user inputs**. Each user input that provides **distinct application oriented data** to the software
  2. **Number of user outputs**. Each user output that provides **application oriented information** to the user is counted. In this context output refers to **reports, screens, error messages, etc.**
  3. **Number of user inquiries**. An inquiry is defined as an **on-line input** that results in the **generation of some immediate software response** in the form of an on-line output.
  4. **Number of files**. Each **logical master file** (i.e., a logical grouping of data that may be one part of a large database or a separate file) is **counted**.
  5. **Number of external interfaces**. All **machine readable interfaces** (e.g., data files on storage media) that are used to **transmit information** to another system are **counted**.

# Product Metrics

## FP calculation

Measurement parameter	Count		Weighting factor			
			Simple	Average	Complex	
Number of user inputs	<input type="text"/>	×	3	4	6	= <input type="text"/>
Number of user outputs	<input type="text"/>	×	4	5	7	= <input type="text"/>
Number of user inquiries	<input type="text"/>	×	3	4	6	= <input type="text"/>
Number of files	<input type="text"/>	×	7	10	15	= <input type="text"/>
Number of external interfaces	<input type="text"/>	×	5	7	10	= <input type="text"/>
Count total						<input type="text"/>

$$FP = \text{Count total} * (0.65 + 0.01 * \sum Fi)$$

# Software Measurement

The  $F_i$  ( $i = 1$  to  $14$ ) are "complexity adjustment values" based on responses to the following questions

1. Does the system require reliable backup and recovery?
2. Are data communications required?
3. Are there distributed processing functions?
4. Is performance critical?
5. Will the system run in an existing, heavily utilized operational environment?
6. Does the system require on-line data entry?
7. Does the on-line data entry require the input transaction to be built over multiple screens or operations?
8. Are the master files updated on-line?
9. Are the inputs, outputs, files, or inquiries complex?
10. Is the internal processing complex?
11. Is the code designed to be reusable?
12. Are conversion and installation included in the design?
13. Is the system designed for multiple installations in different organizations?
14. Is the application designed to facilitate change and ease of use by the user?

• Each of these questions is answered using a scale that ranges from 0 (not important or applicable) to 5 (absolutely essential)

# Software Measurement

Given the following values, **compute function point** when all complexity adjustment factor (CAF) are average and weighting factors are

- User Input = 40 Simple
- User Output = 35 Complex
- User Inquiries = 30 Simple
- User Files = 8 Average
- External Interface = 6 Average

$$\text{Count Total} = 40 * 3 + 35 * 7 + 30 * 3 + 8 * 10 + 6 * 7 = 577$$

$$\Sigma(F_i) = 14 * 3 = 42 \quad (\text{total 14 } F_i \text{ Each is average on scale of 1-5 hence 3})$$

$$\text{FP} = \text{count total} [0.65 + 0.01 \Sigma(F_i)]$$

$$\begin{aligned} \text{FP} &= 577 * (0.65 + 0.42) \\ &= 617.39 \end{aligned}$$

# Software **Measurement**

• Once function points have been calculated, they are used in a manner analogous to LOC as a way to normalize measures for software productivity, quality, and other attributes:

- Errors per FP.
- Defects per FP.
- \$ per FP.
- Pages of documentation per FP.
- FP per person-month.

## **Reference:**

- 1. Rajesh Narang, Software Engineering Principles & Practices, McGraw Hill, (318-329)**
- 2. Roger S. Pressman Software Engineering Practitioner's Approach, McGraw Hill, 5<sup>th</sup> edition (89-91)**

# Software Product Estimation

## COCOMO model (constructive Cost Estimation Model)

2017 Q2 b

- Basic Effort Estimation Model uses size of Code to estimate **man month effort** and **project duration in month**.
- **Effort:** Amount of labour that will be required to complete a task. It is measured in person-months units.
- **Schedule:** Simply means the amount of time required for the completion of the job, which is, of course, proportional to the effort put. It is measured in the units of time such as weeks, months.

# COCOMO

- **Software are categorized into :**

1. **Organic:** If the team size required is adequately small, the problem is well understood and has been solved in the past and also the team members have a nominal experience regarding the problem.

2. **Embedded Projects:** A software project with requiring the highest level of complexity, creativity, and experience requirement fall under this category. Such software requires a larger team size than the other two models and also the developers need to be sufficiently experienced and creative to develop such complex models.

3. **Semi-detached projects:** If the vital characteristics such as team-size, experience, knowledge of the various programming environment lie in between that of organic and Embedded.

The projects classified as Semi-Detached are comparatively less familiar and difficult to develop compared to the organic ones and require more experience and better guidance and creativity. Eg: Compilers or different Embedded Systems can be considered of Semi-Detached type.



# COCOMO

## Coefficients Related to Basic Model

Estimation at the early design stage where ONLY User Requirements are defined

$$\text{Effort} = a * (\text{KLOC})^b$$

$$\text{Project Duration} = c * (\text{Effort})^d \quad (\text{represented in Months})$$

$$\text{Person Required} = \text{Effort} / \text{Project Duration}$$

Development Model	a	b	c	d
Organic	2.4	1.05	2.5	0.38
Embedded	3.6	1.20	2.5	0.32
Semi detached	3.0	1.12	2.5	0.35

# Software Measurement

Suppose a project manager has used FP analysis to estimate lines of source code required to develop a project and finds that the code size is 10,000.

Estimate duration of the project and number of persons required.

a) Assume the project is classified as **Embedded Project**.

b) Assume the project is classified as **Organic Project**.

Development Model	a	b	c	d
Organic	2.4	1.05	2.5	0.38
Embedded	3.6	1.20	2.5	0.32
Semi detached	3.0	1.12	2.5	0.35

$$\text{Effort} = a * (\text{KLOC})^b$$

$$\text{Duration} = c * (\text{Effort})^d$$

$$\text{Person Required} = \frac{\text{Effort}}{\text{Duration}}$$

a) Effort = 57.06 , Project duration = 9.12 months and person =6.26 i.e. 6

b) Effort = 26.92 , Project duration = 8.72 moths and person = 3.08 i.e. 3

# COCOMO II

- Modified to accommodate **Early adjustment factors** and **Scale factors**

$$\text{Effort} = 2.94 * \text{EAF} * (\text{KLOC})^b$$

EAF = Early Adjustment Factors

is multiplication of the seven factors on the scale of Very Low, Low, Normal, High, Very High, & Extra

$$b = 0.91 + (\text{sum of Scale factors})/100$$

# COCOMO II

**EAF = Early Adjustment Cost Factors (7 major):**

## **1. Product Reliability & Complexity**

1. S/w Reliability req. more reliable more time
2. Size of application data : more data size more time
3. Complexity of product : complex will require more time
4. Documentation match: more documentation more time

## **2. Requirement Reusability :**

1. More reusability more development time

# COCOMO II

**EAF = Early Adjustment Cost Factors (7 major):**

## **3.Platform:**

- 1.Execution time constraint: higher constraint more time
- 2.Main storage constraint : higher constraint more time
- 3.Platform volatility (OS/RDBMS/Browser change): More flexibility more time

## **4.Personnel Attribute:**

- 1.Analyst Capability: More ability less time
- 2.Programmer Capability: More ability less time
- 3.Personnel Continuity : More ability less time

# COCOMO II

**EAF = Early Adjustment Cost Factors (7 major):**

## **3. Platform:**

1. Execution time constraint: higher constraint more time
2. Main storage constraint : higher constraint more time
3. Platform volatility (OS/RDBMS/Browser change): More flexibility more time

## **4. Personnel Attribute:**

1. Analyst Capability: More ability less time
2. Programmer Capability: More ability less time
3. Personnel Continuity : More ability less time

# Software **Measurement**

## **5.Previous Experience:**

- 1.Application Experience : More experience less time
- 2.Virtual Machine Experience : More experience less time
- 3.Language & Tool Experience : More experience less time

## **6.Team Support Facilities:**

- 1.Use of Software tools More tools less time
- 2.Multisite: More places more time

## **7.Required development schedule:**

- 1.Normal or compressed : Compressed will take more time

# COCOMO II model

## Scale factors (5 factors):

1. **Precedentness or Previous Experience:** Higher experience lesser time
2. **Development Flexibility:** More freedom to choose convenient development process less time
3. **Risk Resolution:** Lesser analysis done more time
4. **Team Cohesion:** more seamless interaction less time
5. **Process Maturity:** More matured process less time



# 1.4 Project Management Activities: Planning , Scheduling & Tracking

No one plans to fail, but fails to plan

Planning is the **next important** stage in project development

Estimating is as much art as it is science, this important activity need be conducted in a systematic manner.

Useful techniques for time and effort estimation do exist. Process and project metrics can provide historical perspective and powerful input for the generation of quantitative estimates.

Past experience (of all people involved) can aid immeasurably as estimates are developed and reviewed.

Estimation lays a foundation for all other project planning activities and project planning provides the road map for successful software engineering

# Estimation

## Importance of estimation in Planning

- involves estimation an attempt to determine how much money, how much effort, how many resources, and how much time it will take to build a specific software-based system or product
- Performed by Software managers—using information solicited from customers and software engineers and software metrics data collected from past projects
- Projects being costly it is reasonable to develop an estimate before you start creating the software.
- Estimation begins with a description of the scope of the product, decomposed into a set of smaller problems and each of these is estimated using historical data and experience as guides.
- It is advisable to generate your estimates using at least two different methods (as a cross check)
- Problem complexity and risk are considered before a final estimate is made.
- Estimates should attempt to define best case and worst case scenarios so that project outcomes can be bounded.

# Planning & Scheduling

What will be the inputs for planning ? (type in chat box)

- Scope (What to develop)

- Estimation (Time and hence cost)

- Risks associated

- Human Resources available (role, number, when will be available etc.)

# Planning & Scheduling

## Steps in Planning

- prepare a task table mentioning the task name, task id, duration, resource name
- Create a network of these tasks that will enable to get the job done on time

### Some observations:

- In a complex system many tasks may occur in parallel
- Result of work performed during one task may have a profound effect on work to be conducted in another task
- Interdependencies are very difficult to understand without a schedule
- Virtually impossible to assess progress on a moderate or large software project without a detailed schedule.

# Reasons for late delivery of products

- An unrealistic deadline established by someone outside the software development group and forced on managers and practitioners within the group.
- Changing customer requirements that are not reflected in schedule changes.
- An honest underestimate of the amount of effort and/or the number of resources that will be required to do the job.
- Predictable and/or unpredictable risks that were not considered when the project commenced.
- Technical difficulties that could not have been foreseen in advance.
- Human difficulties that could not have been foreseen in advance.
- Miscommunication among project staff that results in delays.
- A failure by project management to recognize that the project is falling behind schedule and a lack of action to correct the problem.

# Late delivery

## Remedies:

- Perform a detailed estimate using historical data from past projects. Determine the estimated effort and duration for the project.
- Using an incremental process model, develop a strategy that will deliver critical functionality by the imposed deadline, but delay other functionality until later. Document the plan.
- Avoid to impose unrealistic deadline

# Project Scheduling

- An activity that distributes estimated effort across the planned project duration by allocating the effort to specific tasks
- specific software tasks are identified and scheduled
- the schedule evolves over time
- During early stages of project planning, a macroscopic schedule is developed identifying all major activities and the product functions to which they are applied.
- As the project gets under way, refine into a detailed schedule.
- Two approaches:
  - Based on the end-date for release distribute effort within the prescribed time frame
  - Based on the time required for each activity, decide the end date

# Basic principle guides of Scheduling

- **Compartmentalization**: Break into a number of manageable activities and tasks by decomposing both the product and the process.

- **Interdependency**: The interdependency of each compartmentalized activity or task must be determined. Some tasks must occur in sequence while others can occur in parallel. Some activities cannot commence until the work product produced by another is available. Other activities can occur independently.

- **Time allocation**: Each task to be scheduled must be allocated some number of work units (e.g., person-days of effort). In addition, each task must be assigned a start date and a completion date that are a function of the interdependencies and whether work will be conducted on a full-time or part-time basis.

- **Effort validation**: Every project has a defined number of staff members. As time allocation occurs, the project manager must ensure that no more than the allocated number of people have been scheduled at any given time. For example, consider a project that has three assigned staff members (e.g., 3 person-days are available per day of assigned efforts). On a given day, seven concurrent tasks must be accomplished. Each task requires 0.50 person days of effort. More effort has been allocated than there are people to do the work.



# Basic principle guides of Scheduling

- **Define responsibilities** Every task that is scheduled should be assigned to a specific team member.
- **Define outcomes** Every task that is scheduled should have a defined outcome. For software projects, the outcome is normally a work product (e.g., the design of a module) or a part of a work product. Work products are often combined in deliverables.
- **Define milestones** Every task or group of tasks should be associated with a project milestone. A milestone is accomplished when one or more work products has been reviewed for quality and has been approved.

# Some facts about Scheduling

- Experienced resources would need less time for development
- Familiar tools & defined process will reduce the development efforts
- **Adding more programmers/ developers will NOT ALWAYS reduce the project duration**
- A recommended distribution (not rules) of effort across the definition and development phases is often referred to as the **40–20–40 rule** (Forty percent of all effort is allocated to **front-end analysis and design** Twenty percent for **coding** and Forty percent applied to **back-end testing**)

# Defining Task set

- Irrespective of the process model used for development, task set that enable a software team to define, develop, and ultimately support computer software.
- A task set is a collection of software engineering work tasks, milestones, and deliverables that must be accomplished to complete a particular project.
- The task set to be chosen must provide enough discipline to achieve high software quality.
- It must **NOT burden** the project team with unnecessary work.
- In order to develop a project schedule, a task set must be distributed on the project timeline.

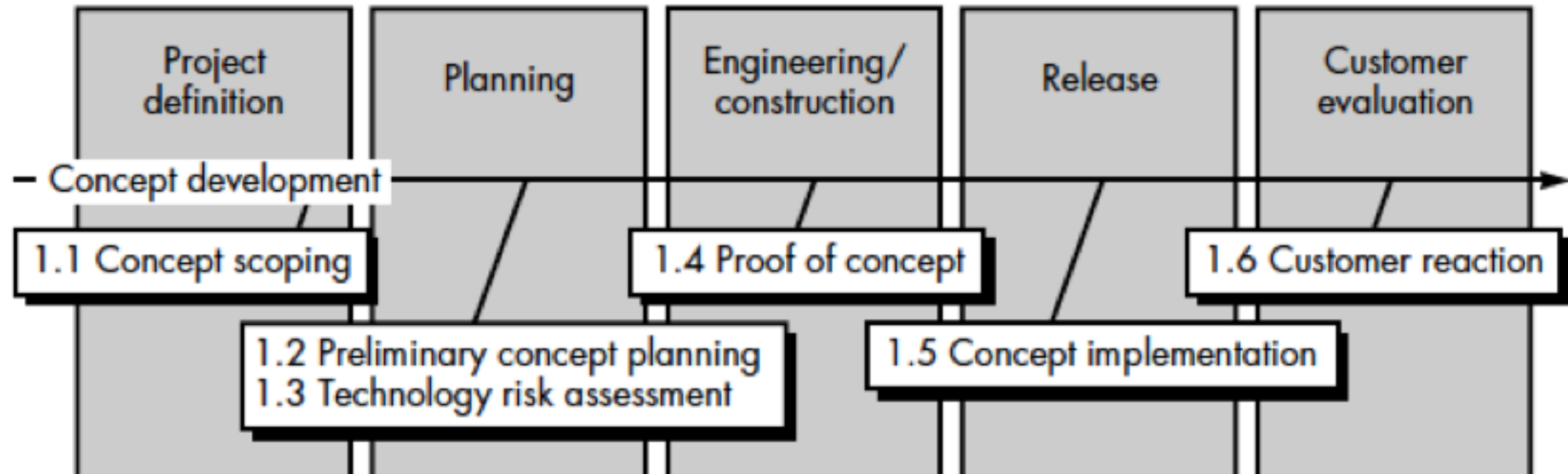
# Defining Task set

- Based on phase of the project identify the various major tasks

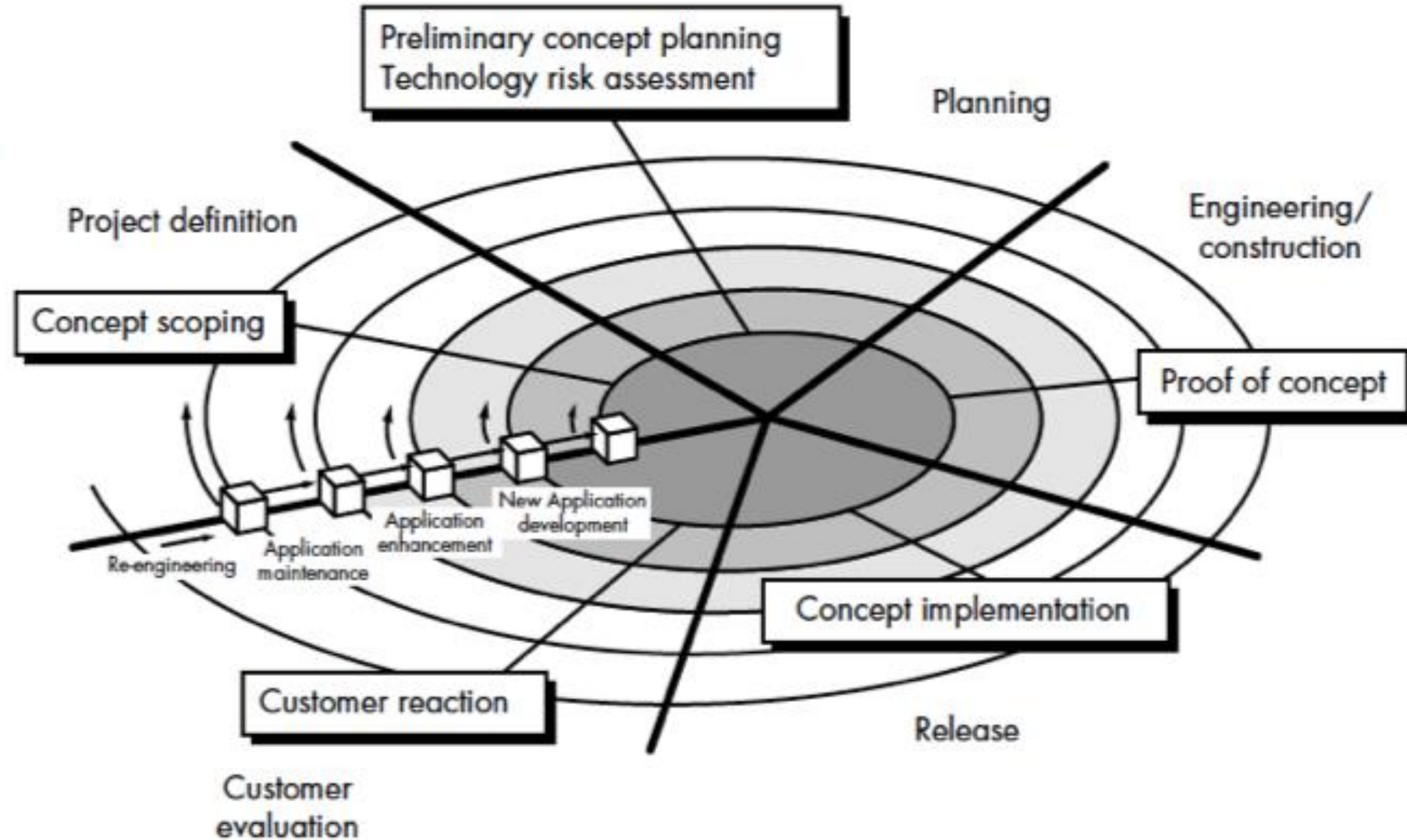
Concept , Analysis , Design, Coding, Maintenance

- Concept scoping determines the overall scope of the project.
- Preliminary concept planning establishes the organization's ability to undertake the work implied by the project scope.
- Technology risk assessment evaluates the risk associated with the technology to be implemented as part of project scope.
- Proof of concept demonstrates the viability of a new technology in the software context.
- Concept implementation implements the concept representation in a manner that can be reviewed by a customer and is used for "marketing" purposes when a concept must be sold to other customers or management.
- Customer reaction to the concept solicits feedback on a new technology concept and targets specific customer applications.

# Sample Linear Development Approach Tasks



# Sample Spiral Development Approach Tasks

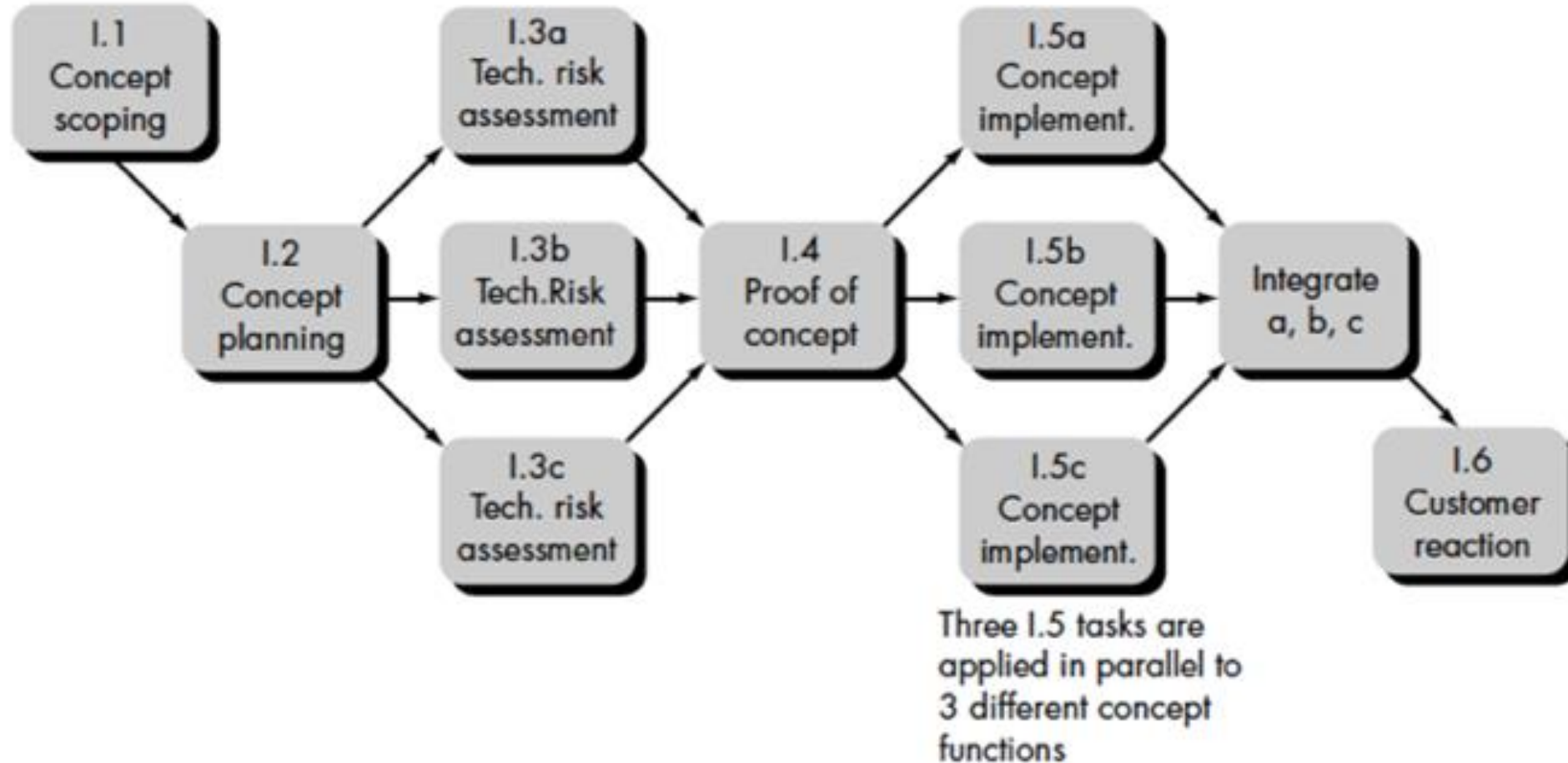


# Sample Task Table

ACTIVITY	DESCRIPTION	IMMEDIATE PREDECESSORS	DURATION (WEEKS)
H	Basic design	—	10
I	Hardware design for A	H	8
J	Hardware design for B	H	6
K	Drawings for B	J	4
L	Software specifications	J	2
M	Parts purchase for B	J	4
N	Parts purchase for A	I	4
O	Drawings for A	I	5
P	Installation drawings	I, J	5
Q	Software purchases	L	5
R	Delivery of parts for B	M	5
S	Delivery of parts for A	N	3
T	Software delivery	Q	3
U	Assembly of A	O, S	1
V	Assembly of B	K, R	5
W	Test A	U	2
X	Test B	V	3
Y	Final installation	P, W, X	8
Z	Final system test	Y, T	6



# Sample Task Network Diagram





# Scheduling

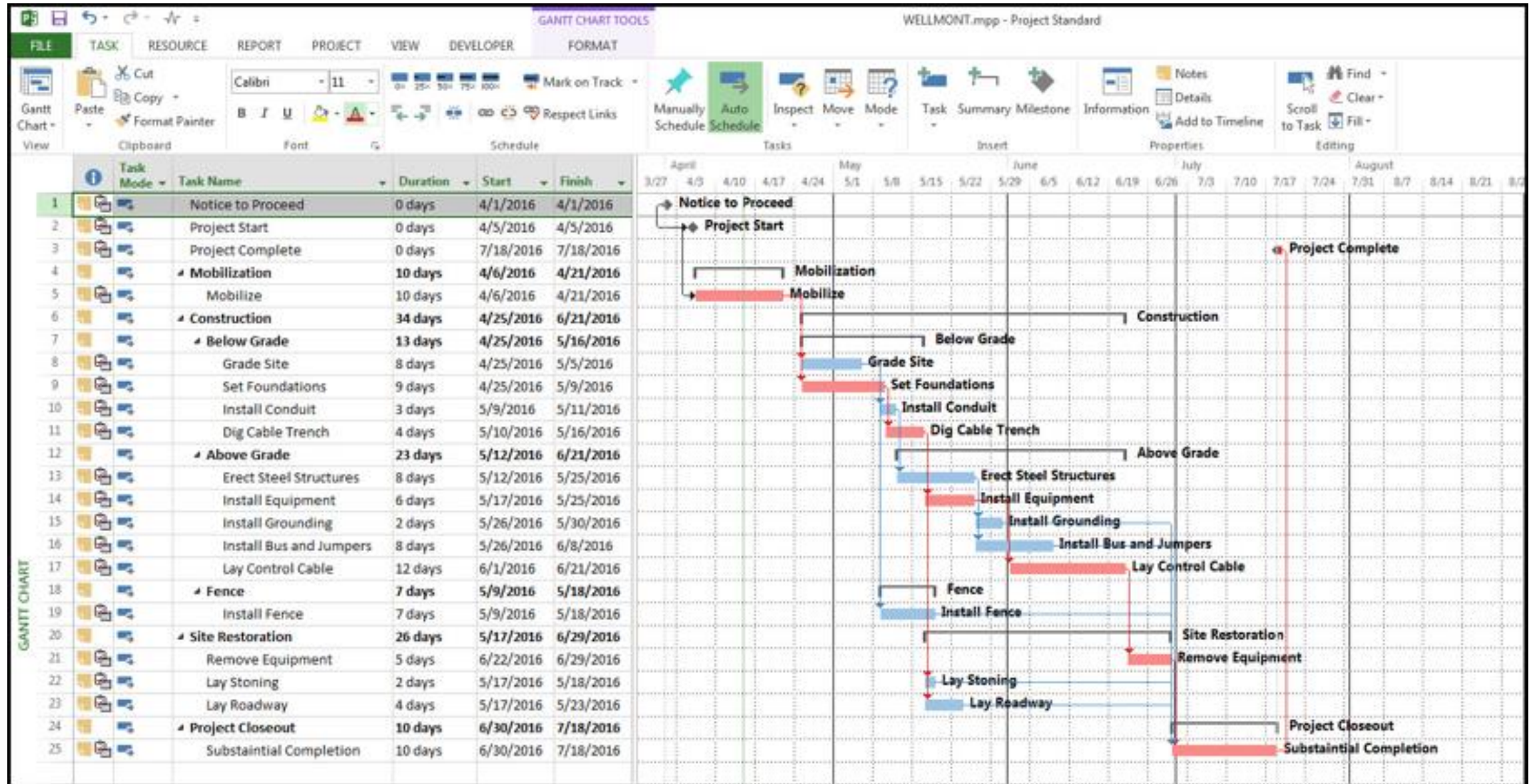
## **Project Scheduling methods:**

- Critical Path Method (CPM)
- Program Evaluation & Review Technique (PERT)

## **Tools:**

- Timeline Chart/ Gantt Chart
- MicroSoft Project Tool used for scheduling

# Scheduling



# Tracking the Schedule

- The project schedule provides a road map
- Properly developed project schedule defines the tasks and milestones that must be tracked and controlled as the project proceeds
- Tracking can be accomplished by:
  - Conducting periodic project status meetings in which each team member reports progress and problems.
  - Evaluating the results of all reviews conducted throughout the software engineering process.
  - Determining whether formal project milestones (the diamonds shown in Figure) have been accomplished by the scheduled date
  - Comparing actual start-date to planned start-date for each project task listed in the resource table
  - Meeting informally with practitioners to obtain their subjective assessment of progress to date and problems on the horizon.
  - Using earned value analysis to assess progress quantitatively.