Unit 1

Introduction to Software Engineering and Software Processes

Introduction

- When the computer industry started in the 1950s no one would have thought that
  - Software would become an indispensable technology for business, science and engineering
  - Software would enable the creation of new technologies (genetic engineering)
  - Software would enable the extension of existing technologies (telecommunications)
  - Software would be responsible for the demise of older technologies (like printing industry)
  - Software would be the driving force behind the PC revolution (Wordstar, Lotus 123)
  - A software company would become larger and more influential than the vast majority of Industrial era companies (GM - Auto , Exxon – Mobil - Oil , Steel)
  - software would slowly evolve from a product to a service as "on-demand" software

Cont...

- companies deliver just-in-time functionality via a Web browser
- A vast software driven network called Internet would evolve and change everything from library search to consumer shopping
- Software would become embedded in systems of all kinds : transportation, medical, telecomm, military, industrial, entertainment, office machines etc.

- It is also a prime example of the law of unintended consequences.
- Millions of programs developed so far would have to be corrected, adapted and enhanced as time passed and that this burden would absorb more people and resources than all work applied to the creation of the new software
- The software development community has continually attempted to develop technologies that will make it easier, faster and less expensive to build and maintain high quality computer software.
- ***"Software Engineering is a framework for the software development community that encompasses a process, a set of methods and an array of tools so that they can do it right."***

THE NATURE OF SOFTWARE

- Today, software takes on a dual role.
- It is both a product and a vehicle for delivering a product.
- It is a product which you can buy (license). As a product it allows you to use the power of hardware.
- It also allows you to deliver the most important product of our time – information.
- Software which is a product creates, stores, transmits, displays information which is again a product.
- As the vehicle used to deliver the product, software acts as the basis for the control of the computer(OS), the communication of information(Networks), the creation and control of other programs (software tools and environment).

- Software delivers the most important product of our time—*information*. It manages business information to enhance competitiveness; It provides a gateway to worldwide information networks (e.g., the Internet), and provides the means for acquiring information in all of its forms.
- The role of computer software has undergone significant change over the last half-century.
- Dramatic improvements in hardware performance, profound changes in computing architectures, vast increases in memory and storage capacity, and a wide variety of exotic input and output options, have all precipitated more sophisticated and complex computer-based systems.

Defining Software

- What is software ?
  - It is a PRODUCT.
- Who builds it ?
  - S/W Engineers design and build it.
- For whom ?
  - For everybody. Because it does affect everybody. Directly or indirectly.
- How?
  - Train reservation. Weather forecast, State/Central budgets, R & D in medicine / pharmaceutical industry.
- Since the software one develops affects everybody directly or indirectly it has to be developed right -meaning by applying engineering approach – systematically and in disciplined way. If you don't then
  - It will not get finished on time
  - There will be cost overrun
  - There will be errors
  - Ultimately it will not meet the basic objective it was supposed to meet.

Cont...

- Software consists of
  - Programs – they provide desired function, features and performance when executed.
  - Data structures that enable programs to manipulate information.
  - Descriptive information in both hard copy and virtual forms that describes the operation and use of the programs

Software Application Domains

- For transportation of goods and people you have number of different types of products like bullock / camel carts , bicycle , scooter / motor cycle , car , bus , truck , train , golf cart , aircraft , boats , ships etc. They all serve different purpose. Similarly there are number of different types of s/w applications which serve different purposes. They are as follows :

1. ***System software*:**
   - System software is a collection of programs written to service other programs.
   - Some system software (e.g., compilers, editors, and file management utilities) process complex, but determinate, information structures.
   - Other systems applications (e.g., operating system components, drivers, telecommunications processors) process largely indeterminate data.
   - They interact with the hardware , have heavy usage by multiple users , have concurrent operation that requires scheduling , resource sharing etc.

Cont...

2. ***Application software***: stand-alone programs that solve a specific business need. Applications in this area process business or technical data in a way that facilitates business operations or management/technical decision making. In addition to conventional data processing applications, application

   software is used to control business functions in real time (e.g., point-of-sale transaction processing, real-time manufacturing process control- Payroll ,invoicing , ERP , Inventory etc.)

3. ***Engineering and scientific*** : has been characterized by "number crunching" algorithms. Applications range from astronomy to volcano logy, from automotive stress analysis to space shuttle orbital dynamics, and from molecular biology to automated manufacturing. Computer-aided design, system simulation, and other interactive applications have begun to take on real-time and even system software characteristics.

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.. Microwave ovens , washing machines , dishwashers , cell phones.
5. ***Product Line software*** : designed to provide a specific capability for use by many different customers. Product-line software can focus on a limited and esoteric marketplace (e.g., inventory control products) or address mass consumer markets (e.g., word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, and personal and business financial applications).

6. ***Web based software*** : called "WebApps," this network-centric software category spans a wide array of applications. In their simplest form, WebApps can be little more than a set of linked hypertext files that present information using text and limited graphics. However, as Web 2.0 emerges, WebApps are evolving into sophisticated computing environments that not only provide stand-alone features, computing functions, and content to the end user, but also are integrated with corporate databases and business applications.
7. ***Artificial intelligence software*** : makes use of nonnumerical 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.

New Challenges

- Software engineers are facing lot of challenges related to legacy software – software developed few years ago.
- But, there are new challenges which have appeared on the horizon. Some of these are :
  - Open world computing
    - The rapid growth of wireless computing may lead to distributed computing. You do not need to be connected to the network by wires.
    - Small devices (even mobile phones) will be able to get connected to the network and the challenge will be to develop software useful on such devices.
  - Netsourcing
    - Developing applications to run on WWW which can be used by end users all over the world.

Cont...

- Open Source where the source code is available for customers to make changes. The challenge is to build source code that is self descriptive.
- New Economy
  - The economy based on ICT (information and communication technology) is called the new economy as compared to the manufacturing based economy which is now called old economy.
  - The new economy was at peak in 1990s but then the bubble burst in the early 2000.
  - Lot of people thought that the new economy is dead but once again we can see that it is well and alive and is evolving (Internet 2.0, new apps etc).
  - The challenge is to develop applications that will allow mass communication and distribution using concepts which are evolving now.

Legacy Software

- ***Software programs developed few years ago are* called legacy software**.
- "Legacy software systems . . . were developed decades ago and have been continually modified to meet changes in business requirements and computing platforms. The proliferation of such systems is causing headaches for large organizations who find them costly to maintain and risky to evolve."
- These software programs have been modified number of times over a period of number of years since they are critical for the business.
- Because of this criticality they have been in use for a long period.
- Many of them have one characteristic in common and that is poor quality.

- Some of the reasons for the poor quality are
  - inextensible design,
  - poor or nonexistent documentation,
  - poorly managed change history etc.
- As long as they serve their main purpose they should be used but then they will need to evolve to take care of changes in technology and changes in business requirements.
- You have to reengineer the system to keep it viable into the future. (Development of entirely new software will most of the times be very expensive).
- So the goal of modern software engineering is to “device methodologies that are founded on the notion of evolution”.

Software Myths (erroneous beliefs)

- Software projects still have cost overruns , time overruns , high failure rate , low acceptance by users.
- Why ? Because people do not use engineering approach and still believe in and do things which are not right or proper.
- They are called myths.
- Myths can be classified as Management myths , customer myths , practitioner's myths etc.

- Management Myths
  1. *We have standards and rules so my team will be able to develop good s/w.* But , are the people aware about the rules and standards? If they are , are they used ? Are they complete? Answers to most of these questions in lot of cases are NO.
  2. *We have latest hardware and hence our s/w will be developed fast.* What you need is CASE tools and use of them. Fast hardware does not ensure fast development.
  3. *If we fall behind we will catch up by adding more programmers.* Adding more programmers will actually delay the project. Training to new people , more communication , more monitoring and coordination.
  4. *If I decide to outsource the software project to a third party, I can just relax and let that firm build it.* If an organization does not understand how to manage and control software projects internally, it will invariably struggle when it outsources software projects.

- Customer Myths
  1. *General statement of objectives is sufficient to start programming.* We know that programming can not start until most of the requirements are understood and the system has been designed.
  2. *Project requirements continuously change but change can be accommodated because the s/w is flexible*. Time is crucial. When is the change required ? Earlier in the life cycle of the project better.

- Practitioner's myths
  1. *Once program is written and it works my job is done. So I should finish the program fast*. In reality maintenance of a system takes up 60-80 % of the total time spent on the system. So faster completion of a working program should not be the objective.
  2. *Quality can be assessed only after completing the program*. No , actually formal technical reviews during development can help.
  3. *Working program is the only deliverable*. A working program is only one part of a software configuration that includes many elements. A variety of work products (e.g., models, documents, plans) provide a foundation for successful engineering and, more important, guidance for software support.
  4. Soft*ware engineering will make us create voluminous and unnecessary documentation and will invariably slow us down*. No , actually software engineering is about quality. Better quality results in reduced overall work, faster delivery and more satisfaction on the part of user.

Unique Nature of WEBAPPS

- In the early days of the WWW, websites consisted of little more than a set of linked hypertext files that presented information using text and limited graphics.
- As time passed, the augmentation of HTML by development tools (e.g., XML, Java) enabled web engineers to provide computing capability along with informational content.
- web-based systems and applications collectively called as webApps were born.
- Today, webApps have evolved into sophisticated computing tools that not only provide stand-alone function to the end user, but also have been integrated with corporate databases and business applications.
- The following attributes are encountered in the vast majority of WebApps:

1. Network intensiveness.
   - A webApp resides on a network and must serve the needs of a diverse community of clients. The network may enable worldwide access and communication (i.e., the Internet) or more limited access and communication (i.e., a corporate Intranet).
2. Concurrency.
   - A large number of users may access the WebApp at one time.
3. Unpredictable load.
   - The number of users of the WebApp may vary by orders of magnitude from day to day.
4. Performance.
   - If a webApp user must wait too long (for access, for serverside processing, for client-side formatting and display), he or she may decide to go elsewhere.

5. Availability.
   - Although expectation of 100 percent availability is unreasonable, users of popular webApps often demand access on a 24 / 7 / 365 basis.
6. Data driven.
   - The primary function of many webApps is to use hypermedia to present text, graphics, audio, and video content to the end user. In addition, webApps are commonly used to access information that exists on databases that are not an integral part of the Web-based environment (e.g., e-commerce or financial appiications).
7. content sensitive.
   - The quality and aesthetic nature of content remains an important determinant of the quality of a WebApp

8. continuous evolution.
   - Unlike conventional application software that evolves over a series of planned, chronologically spaced releases, web applications evolve continuously. It is not unusual for some webApps (specifically, their content) to be updated on a minute-by-minute schedule or for content to be independently computed for each request.
9. Immediacy.
   - Although immediacy -the compelling need to get software to market quickly-is a characteristic of many application domains, WebApps often exhibit a time-to-market that can be a matter of a few days or weeks.
10. Security.
    - Because WebApps are available via network access, it is difficult, if not impossible, to limit the population of end users who may access the application. In order to protect sensitive content and provide secure modes of data transmission, strong security measures must be implemented throughout the infrastructure that supports a webApp and within the application itself.

11. Aesthetics.

- An undeniable part of the appeal of a webApp is its look and feel. w hen an application has been designed to market or sell products or ideas ,aesthetics may have as much to do with success as technical design

Software Engineering

- In order to build software that is ready to meet the challenges of the twenty-first century, you must recognize a few simple realities:
  - Problem should be understood before software solution is developed
    - As everybody has different opinions
  - Design is a pivotal activity
  - Software should exhibit high quality
  - Software should be maintainable
- These realities lead to one conclusion: *software in all of its forms and across all of its application domains should be engineered. And that leads us to the subject—software engineering*
- Definition
  - Software engineering is the establishment of sound engineering principles in order to obtain reliable and efficient software in an economical manner.
  - Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software.

- Software engineering is a layered technology. Referring to Figure any engineering approach (including software engineering) must rest on an organizational commitment to quality.
- You can look at software engineering from different angles but the main focus is on quality.
  - If the quality of software is not good, the customers would not be satisfied and your efforts would be wasted.
- Software engineering can be seen as consisting of different layers. There are 3 layers.
  - Processes ,
  - Methods and
  - Tools.

FIGURE 2.1
Software engineering layers

- The first one is the layer of processes / activities.
  - The activities or processes are also called Key Process Areas (KPAs).
  - Some of them are Software configuration management , s/w quality assurance , s/w project planning , requirements management , intergroup coordination , training program , s/w quality management.
- The second layer is software methods.
  - For doing every activity there are methods – How to do the activity.
  - For example – Requirements analysis is an activity or KPA .
  - You can gather requirement by different methods e.g. group interview , personal interview , questionnaire (printed – Web) , descriptive writing etc.

- The third layer is that of tools.
  - They provide automated or semi-automated support for the process and methods.
  - One of the activities is Analysis and design.
  - Methods are DFD and ERD or UML diagrams. There are tools available which help you in preparing them and even generate code.
  - They are called tools. CASE tools (Computer Aided Software Engineering tools)

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 encompasses a set of tasks that produce a major work product(e.g .,an architectural design model).
- A task focuses on a small, but well-defined objective(e.g. conducting a unit test) that produces a tangible outcome.
- A process framework establishes the foundation for a complete software engineering process by identifying a small number of framework activities that are applicable to all software projects, regardless of their size or complexity.
- In addition, the process framework encompasses a set of *umbrella activities that are applicable across the entire software process*.

Generic process framework

- A generic process framework for software engineering encompasses five activities.
- Communication:
  - Before any technical work can start, it is critically important to communicate and collaborate with the customer.
  - The intent is to understand stakeholders, objectives for the project and to gather requirements that help define software features and functions.
- Planning
  - Any complicated journey can be simplified if a map exists.
  - A software project is a complicated journey, and the planning activity creates a "map" that helps guide the team as it makes the journey.
  - The map called a *software project plan*-defines the software engineering work by describing
    - the technical tasks to be conducted,
    - the risks that are likely,
    - the resources that will be required,
    - the work products to be produced, and
    - a work schedule.

- Modeling
  - Whether you're a landscaper, a bridge builder, an aeronautical engineer, a carpenter, or an architect, you work with models every day.
  - You create a "sketch" of the thing so that you'll understand the big picture.
  - If required, you refine the sketch into greater and greater detail in an effort to better understand the problem and how you're going to solve it.
  - A software engineer does the same thing by creating models to better understand software requirements and the design that will achieve those requirements.
- Construction
  - This activity combines code generation (either manual or automated) and the testing that is required to uncover errors in the code.

- Deployment
  - The software (as a complete entity or as a partially completed increment) is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation.
- These five generic framework activities can be used during the development of small, simple programs, the creation of large web applications, and for the engineering of large, complex computer-based systems.
- The details of the software process will be quite different in each case, but the framework activities remain the same.

Software Engineering Umbrella Activities

- Software engineering process framework activities are complemented by a number of umbrella activities.
- In general, umbrella activities are applied throughout a software project and help a software team manage and control progress, quality, change, and risk.
- Typical umbrella activities include
- Software project tracking and control
  - allows the software team to assess progress against the project plan and take any necessary action to maintain the schedule.
- Risk management
  - assesses risks that may affect the outcome of the project or the quality of the product.

- Software quality assurance
  - defines and conducts the activities required to maintain software quality
- Technical reviews
  - Assesses software engineering work products in an effort to uncover and remove errors before they are propagated to the next activity
- Measurement
  - define and collect process, project, and product measures to assist team in delivering software meeting customer needs
- Software configuration management
  - manages the effects of change throughout the software process

- Reusability management
  - defines criteria for work product reuse and establish mechanisms to achieve component reuse
- Work product preparation and production
  - activities to create models, documents, logs, forms, etc.

Software Practice Core Principles

- Principle is an underlying law or assumption that guides you in your work.
- There are some principles which are common to all software engineering practices and some principles which are specific to a particular practice.
- Here we will see the core principles – the principles which are common to all software engineering practices.
- There are 7 such core principles
  1. Understand the reason why the software system exists.
  2. Keep it Simple
  3. Maintain the vision
  4. What you produce others will consume
  5. Be open to future
  6. Plan ahead for reuse
  7. Think
- These are known as ***Hooker's seven principal.***

1. The reason a software system exists is to provide value to the users. If something does not add value to the system do not do it.
2. Keep it Simple Do whatever is necessary but do not complicate things unnecessarily. Keeping the things simple makes the software more maintainable and less error-prone.
3. Maintain the vision Do not lose sight of the ultimate goal.
4. What you produce others will use , maintain or document so do your job in such a way that their job becomes easy.
5. Be open to future. Make your system adaptable (flexible, adjustable) to changes because changes will come.

6. Plan ahead for reuse.

- The previous point was regarding adaptability of this particular software to take care of changes.
- This point is regarding reuse of whole or part of this software in other future software projects.
- Reuse in a project saves time and effort in that project.
- But designing and developing for reuse requires thought and planning in this project.
- It may appear that it will take more time and effort in this project but reuse in future projects will ultimately save overall time and effort of your organization.

7. Think.

- This principle is probably the most overlooked (ignored).
- Placing clear, complete thought before action almost always produces better results.
- You can do things without thinking or after giving proper thought.
- When you think about something you are more likely to do it right.
- If you do think about something and still do it wrong, it becomes a valuable experience.
- When clear thought has gone into a system, value comes out.

SAFE HOME EXAMPLE

How a Project Starts

The scene: Meeting room at CPI Corporation, a (fictional) company that makes consumer products for home and commercial use.

The players: Mal Golden, senior manager, product development; Lisa Perez, marketing manager; Lee Warren, engineering manager; Joe Camalleri, executive VP, business development

The conversation:

Joe: Okay, Lee, what's this I hear about your folks developing a what? A generic universal wireless box?

Lee: It's pretty cool . . . about the size of a small matchbook . . . we can attach it to sensors of all kinds, a digital camera, just about anything. Using the 802.11g wireless protocol. It allows us to access the device's output without wires. We think it'll lead to a whole new generation of products.

Joe: You agree, Mal?

Mal: I do. In fact, with sales as flat as they've been this year, we need something new. Lisa and I have been doing a little market research, and we think we've got a line of products that could be big.

Joe: How big . . . bottom line big?

Mal (avoiding a direct commitment): Tell him about our idea, Lisa.

Lisa: It's a whole new generation of what we call "home management products." We call 'em *SafeHome*. They use the new wireless interface, provide homeowners or small-business people with a system that's controlled by their PC—home security, home surveillance, appliance and device control—you know, turn down the home air conditioner while you're driving home, that sort of thing.

Lee (jumping in): Engineering's done a technical feasibility study of this idea, Joe. It's doable at low manufacturing cost. Most hardware is off-the-shelf. Software is an issue, but it's nothing that we can't do.

Joe: Interesting. Now, I asked about the bottom line.

Mal: PCs have penetrated over 70 percent of all households in the USA. If we could price this thing right, it could be a killer-App. Nobody else has our wireless box . . . it's proprietary. We'll have a 2-year jump on the competition. Revenue? Maybe as much as 30 to 40 million dollars in the second year.

Joe (smiling): Let's take this to the next level. I'm interested.

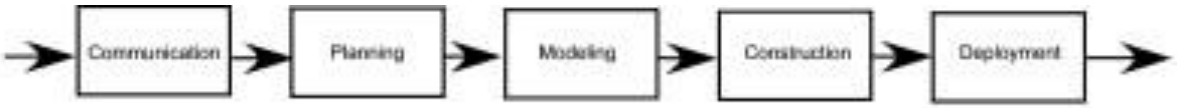
Chapter 2 Process Models

- ***Generic Process Model:***
- A process was defined as a collection of work activities, actions and tasks that are performed when some work product is to be created.
- Each of these activities, actions and tasks reside within a framework or model that defines their relationship with the process and with one another.
- The software process is represented schematically as follows :
  - There are framework activities.
  - Activities have actions.
  - Actions have tasks.
- A generic process framework for software engineering defines five framework activities – Communication, Planning, modeling, Construction and Deployment. Then, there is a set of umbrella activities.

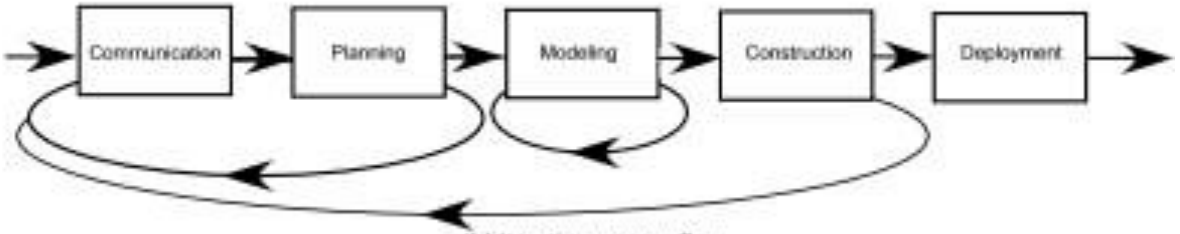
- These activities have to be carried out in some sequence. That sequence is called process flow. There are different types of process flows as shown below :

  Linear, Iterative, Evolutionary and Parallel

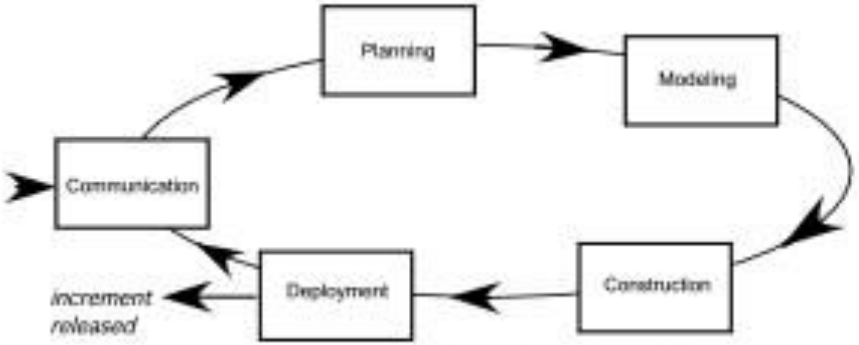
- *Linear process flow* executes each of the framework activities in order beginning with communication and ending with deployment
- *Iterative process flow* executes the activities in a circular manner creating a more complete version of the software with each circuit or iteration
- An *evolutionary process flow* executes the activities in a "circular manner". Each circuit through the five activities leads to a more complete version of the software
- *Parallel process flow* executes one on more activities in parallel with other activities



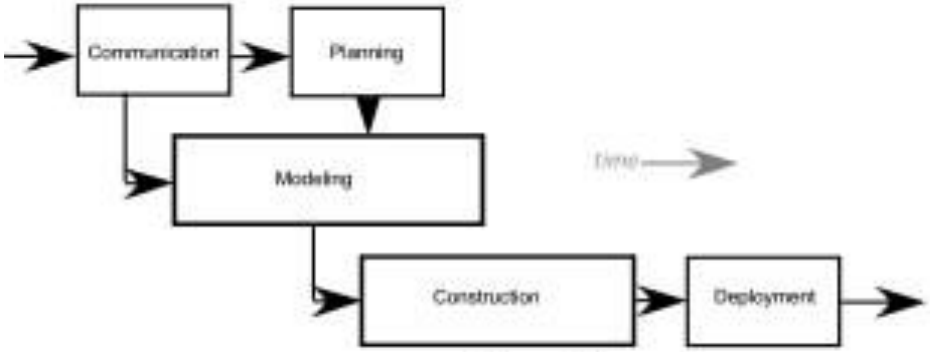
(a) linear process flow



(b) iterative process flow



(c) evolutionary process flow



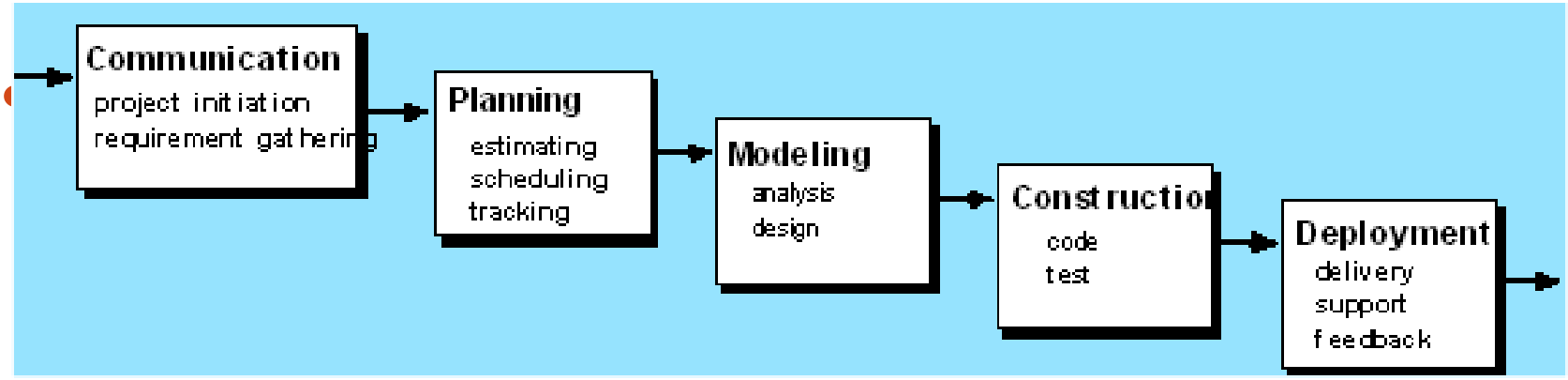
(d) parallel process flow

Prescriptive Process Models

- Earlier, the software development was chaos (mess, confusion, disorder).
- There was no software process.
- Software development was considered to be an art – not a science (no rules).
- Then, prescriptive process models were proposed to bring order to the chaos. These models are called prescriptive because they prescribe what should be done (steps to be carried out to develop software).
- These models brought a certain amount of useful structure to software engineering work. Some of these models are as follows :
  - Waterfall model
  - Incremental Process Models
  - Evolutionary process Models – Prototyping, Spiral
  - Concurrent Models
  - Specialized Process Models
    - Component Based Development
    - The Formal Methods Model
    - Aspect Oriented S/W Development

Waterfall Model

- It is also known as classic life cycle or linear sequential model.
- It suggests a systematic, sequential approach.
- All activities are done one after another only after its predecessor is over.
- One can not go back to previous activity.
- Water once poured down a staircase can not go up to the higher step.



- Following assumptions are made in this model
  - The user must give his / her complete requirements in advance and in detail. This is difficult in real life.
- Completed system will be delivered after the linear sequence is complete.
- Since the working version is available almost at the end of the time span the user must have lot of patience.
- Another problem is that a major blunder – if undetected – can be disastrous.

Summary

- Requirements are well specified, well-defined, well-understood.
- Work-flow is linear
- Sometimes called Classic Life Cycle
- Sequential systematic approach

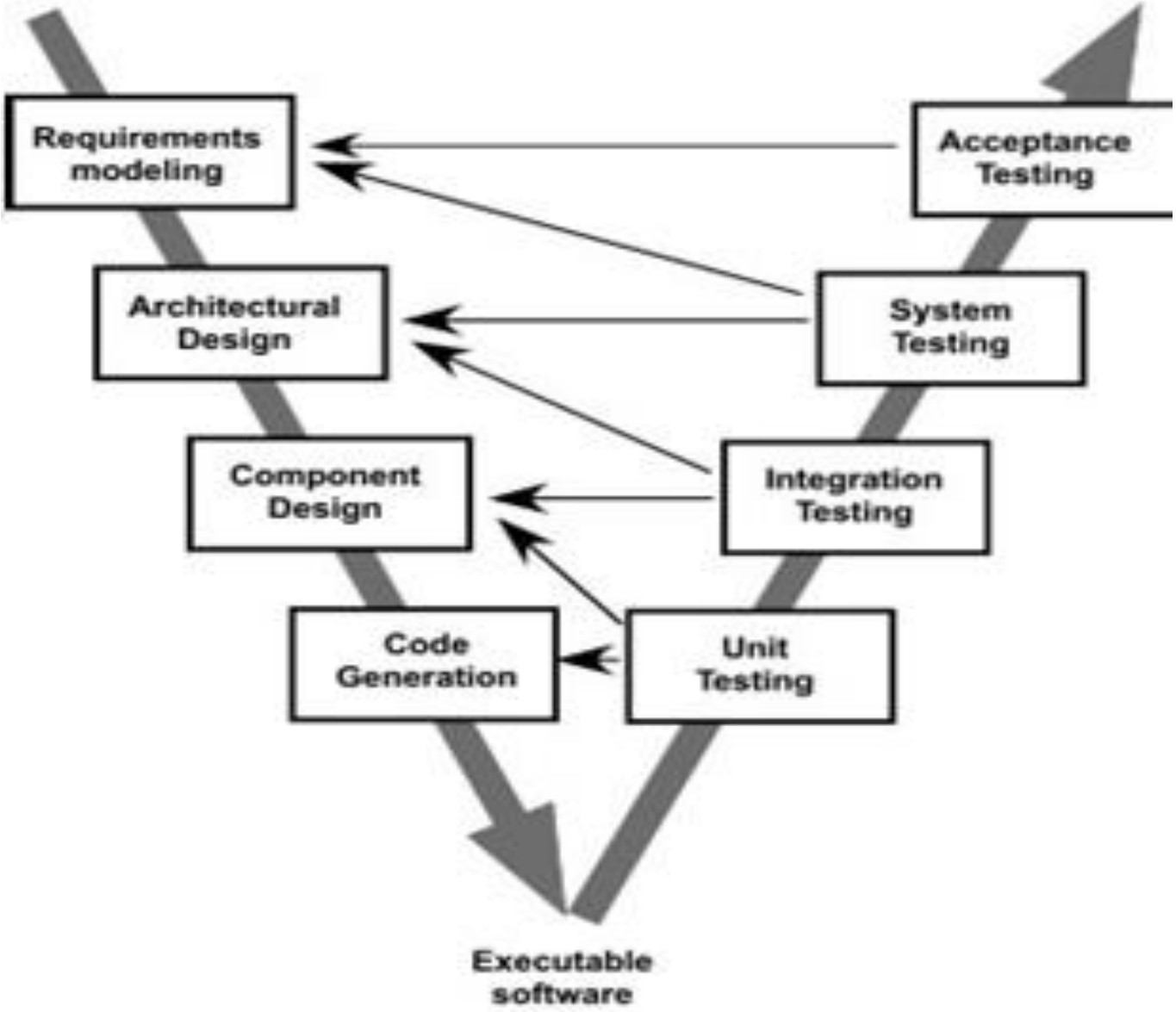
E.g.:

An adaptation in existing accounting s/w because of govt. changes.

- Disadvantages of WFM
  - Real projects rarely follow sequential flow .
  - Difficult to state all requirements explicitly.
  - Customer must have patience.
  - Time consuming.
  - Some team have to wait for another

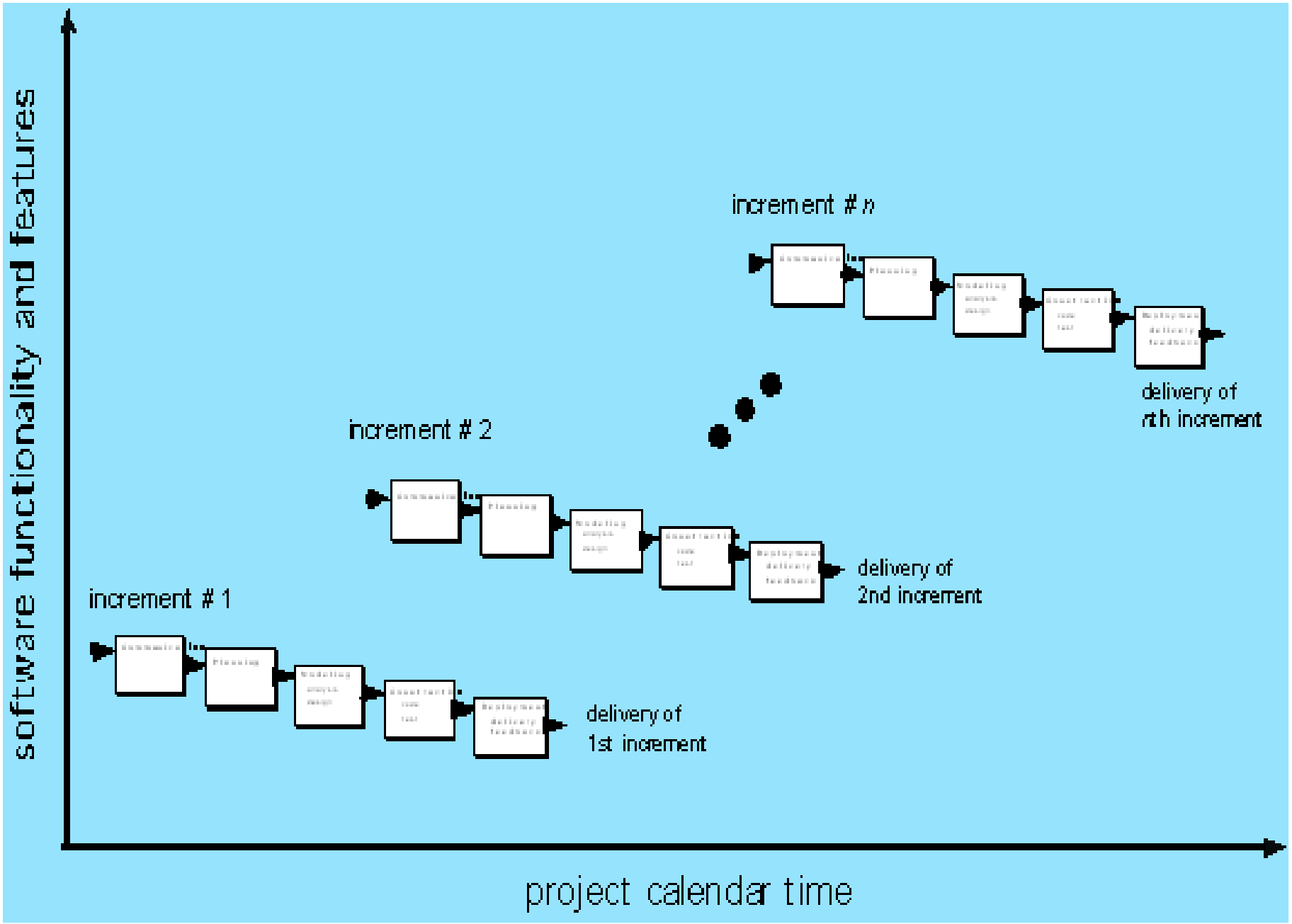
V-Model

- A variation in the representation of the waterfall model is called V-model.
- It depicts the relationship of quality assurance actions to the actions associated with communication, modeling and early construction activities.
- Quality is very important.
- In spite of its limitations this model is better than a haphazard approach.
- Basic requirements are refined into more details when moves down left side of V.
- Performing series of tests that validate each of the models.


Requirements modeling
Acceptance Testing
Architectural Design
System Testing
Component Design
Integration Testing
Code Generation
Unit Testing
Executable software

Incremental Process Models

- It combines elements of linear sequential model with the iterative philosophy of prototyping.
- First we develop a core product and deliver it.
- It will have the basic functionality. We follow steps of analysis , design , development , testing.
- Then , based on the feedback we improve existing features and add some new features.
- For this also we follow the 4 basic steps.
- Example : development of a word processor. Each increment is an operational product and a functional subset of the final product.
- This model can be used when complete staffing is not available (or is not wanted) in the beginning.
- The core product can be developed by a core team and the team for the successive versions can be bigger depending on the success of the product. Focus is on delivery of an operational product.


software functionality and features
project calendar time
increment # 1
delivery of
1st increment
increment # 2
delivery of
2nd increment
increment # n
delivery of
nth increment

Summary

- Initial s/w requirements are well defined but over all scope of the development is not purely known.
- Produces the s/w in increments.
- Linier sequence in staggered fashion.
- First increment is core product delivered to the customer.
- E.g.:

  Word processing s/w
- Advantages of incremental model
  - Can be used when staffing is unavailable.
  - Customer satisfaction.
  - Implements can be planned to manage technical risks.
  - If particular h/w is unavailable; we plan for that development later

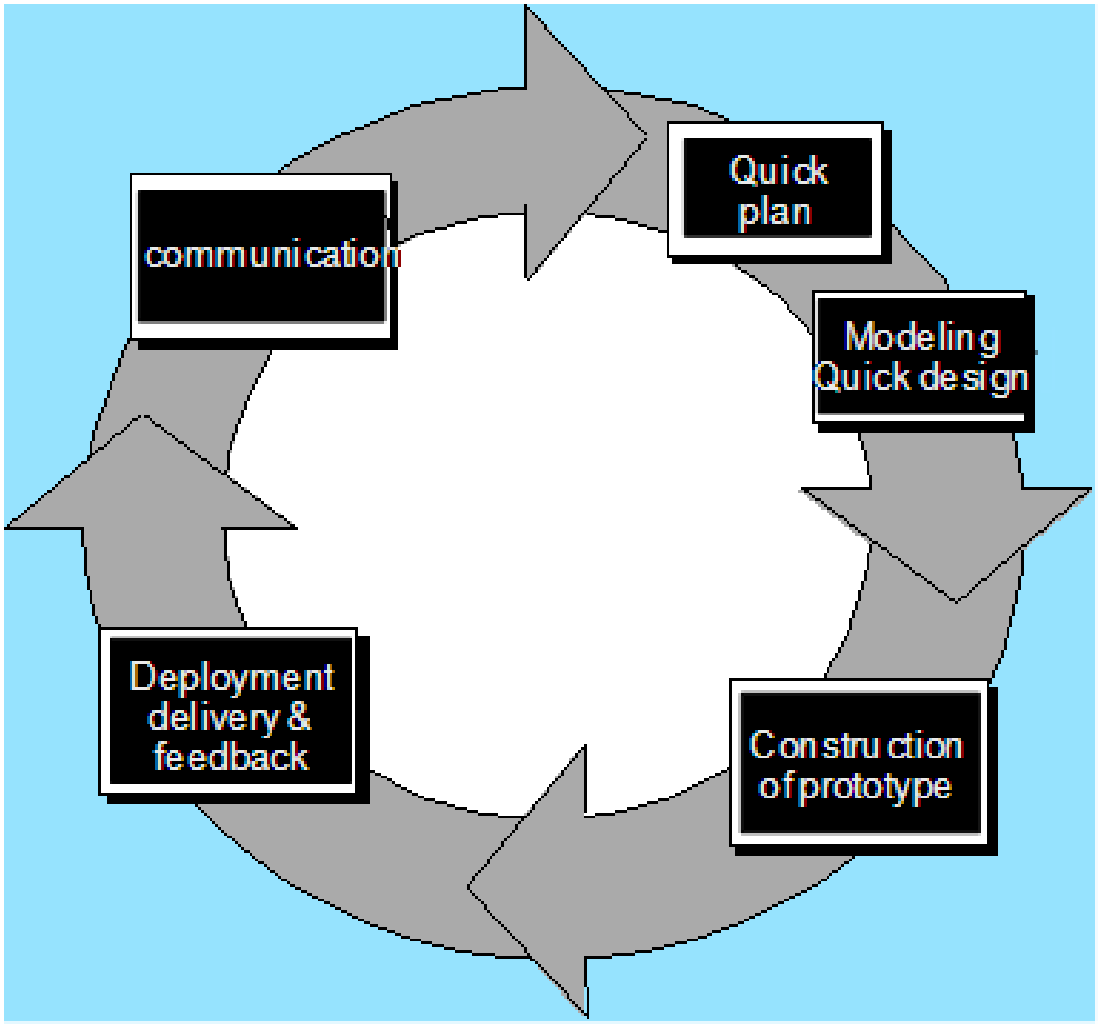
Evolutionary Process Models

- They are iterative. They enable the s/w developer to develop increasingly more complex versions of the software. Why do you require evolutionary models?
- Like all complex systems s/w evolves over a period of time and hence evolutionary models are more suited to s/w development.
- Requirements change while s/w gets developed.
- Market requirements may dictate that.
- There are two evolutionary process models -
  - Prototyping and
  - Spiral.

Prototyping model

- It starts with requirements gathering.
- Overall objectives are defined, known requirements are identified , area where further details are required are identified and a quick design is done.
- The main focus is on understanding user inputs and outputs.
- A prototype (example , sample , model ,trial product) is prepared , evaluated by the customer and changes are suggested.
- This process is done in number of iterations so as to define the requirements. Ideally the prototype should be discarded once the requirements are clear.
- Why ? Because the main objective of this model is to understand the user requirements clearly and then develop the actual software.

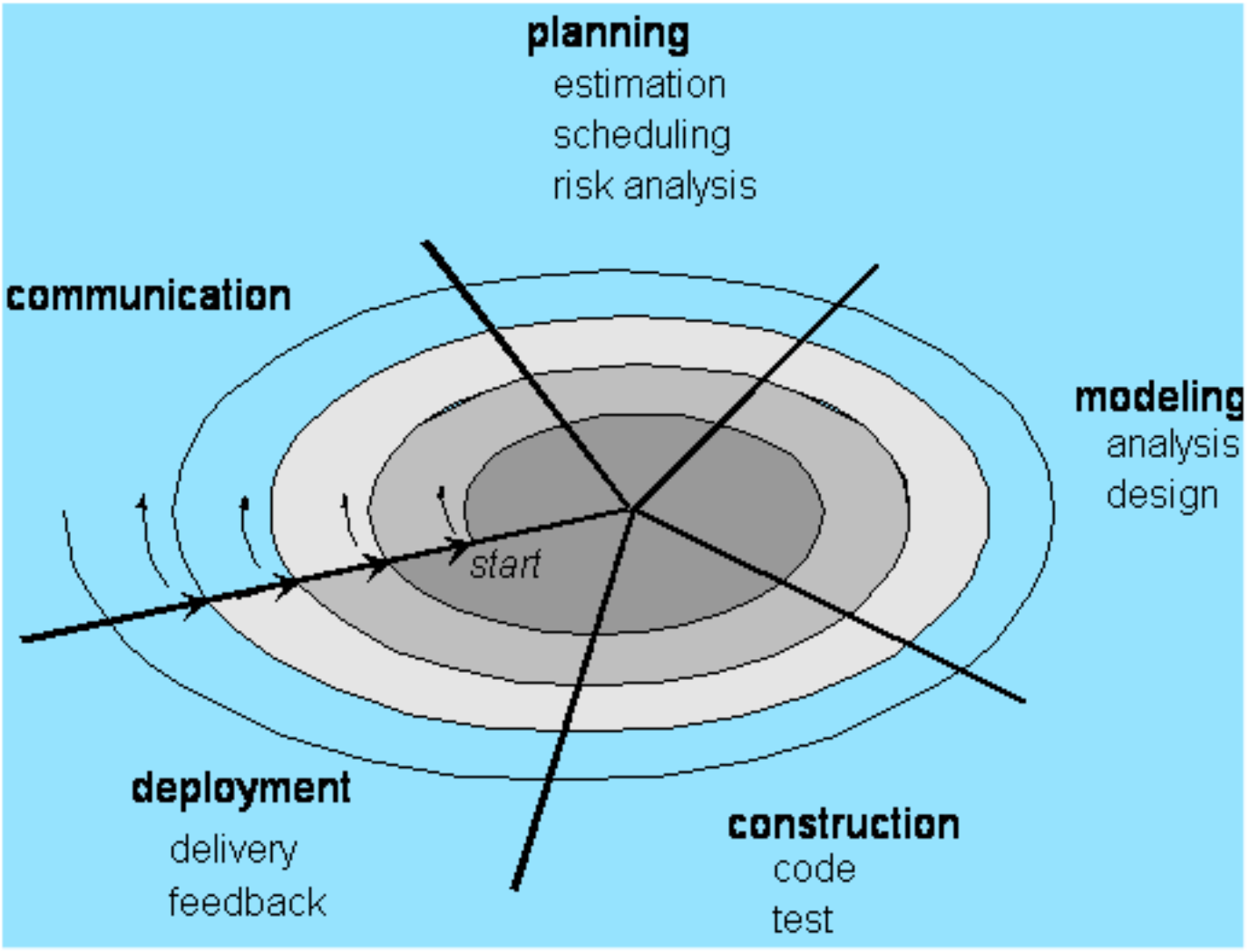
- In that process the s/w developer might have made some compromises regarding quality, performance and maintainability.
- (Instead of using Oracle / SQL server we might have used MS Access as database – which may not be ideal for the real application or used not so efficient algorithm to show only functionality).
- Therefore , the rules of the game should be made clear to the customer in the beginning when you go for prototyping model.
- The idea is to get the requirements and even though the s/w may appear to be ideal it will be discarded once it serves its purpose.


communication
Quick plan
Modeling Quick design
Construction of prototype
Deployment delivery & feedback

The Spiral Model

- This model is useful for large scale systems (like development of OS).
- It is evolutionary model (monkey → man) like incremental model. It was proposed by Barry Boehm.
- This model also combines iterative approach of prototyping model with systematic approach of the linear sequential model.
- Intended for large, expensive and complicated projects.
- It allows for incremental releases of the product, or incremental refinement through each time around the spiral.
- The spiral model also explicitly includes risk management within software development.
- Identifying major risks, both technical and managerial, and determining how to lessen the risk helps keep the software development process under control.
- At each iteration around the cycle, the products are extensions of an earlier product.

- Starting at the center, each turn around the spiral goes through several task regions :
  - Determine the objectives, alternatives, and constraints on the new iteration.
  - Evaluate alternatives and identify and resolve risk issues.
  - Develop and verify the product for this iteration.
  - Plan the next iteration.
- These task regions are performed for each stage of development e.g. concept development (product specs) , new development project , product enhancement , product maintenance.
- Iteration is there at every stage of product development.
- This model more realistically reflects the real world but still is not widely used. Development of OS.


planning
estimation
scheduling
risk analysis
communication
modeling
analysis
design
start
deployment
delivery
feedback
construction
code
test

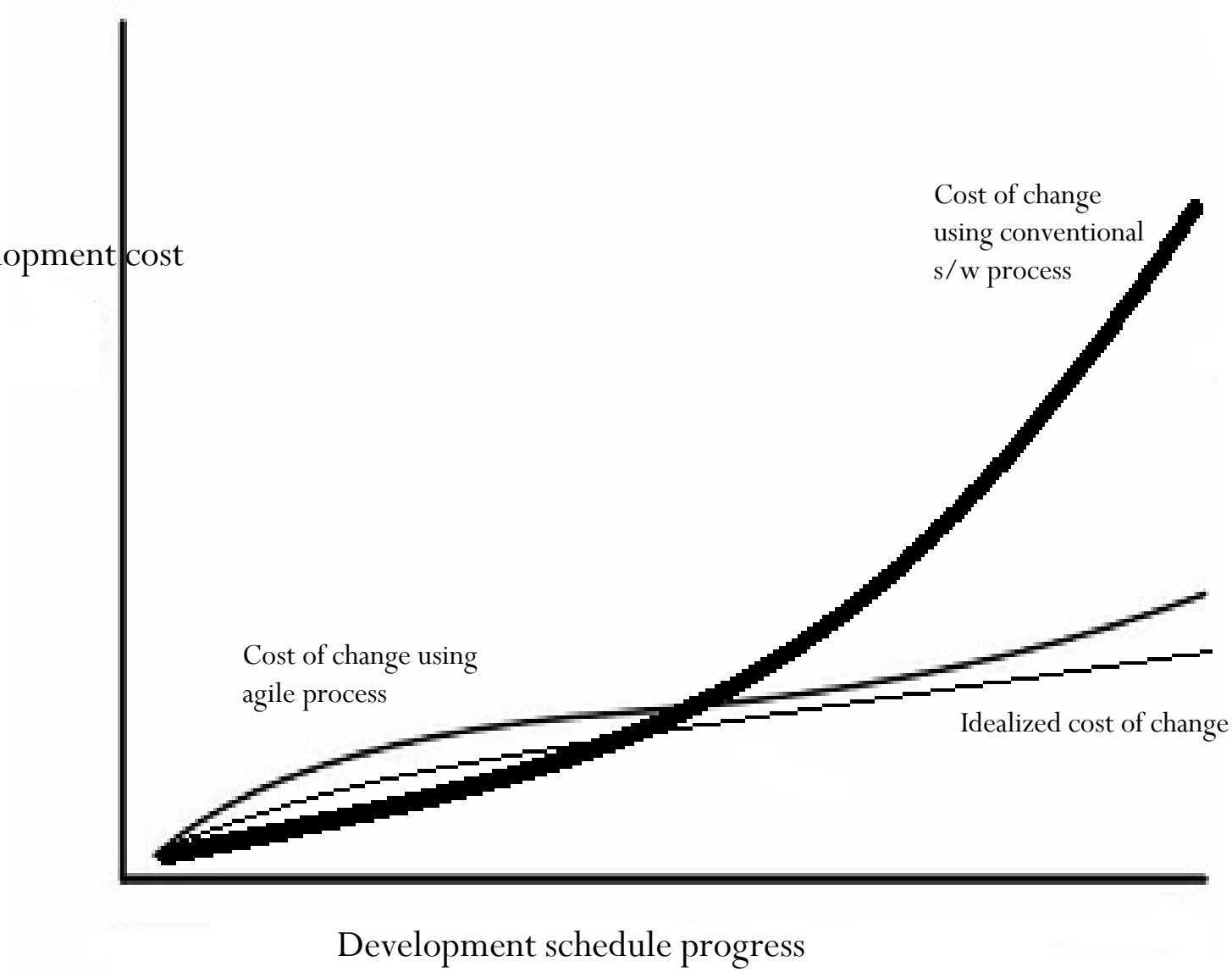
Chapter 3: Agile Development

- Manifesto for "Agile Software Development" states :
- We are uncovering better ways of developing software and helping others do it. Through this work we have come to value:
  - Individuals and interactions over processes and tools
  - Working software over comprehensive documentation
  - Customer collaboration over contract negotiation
  - Responding to change over following a plan
- That is, while there is value in the items on the right, we value the items on the left more.
- Agile methods were developed in an effort to overcome perceived and actual weaknesses in conventional software engineering.
- Too much time spent on planning, modeling, documentation etc but ultimately the customer is not happy.

- Agile development can provide important benefits, but it is not applicable to all projects, all products, all people, and all situations.
- It is also not (opposing) to solid software engineering practice and can be applied as an overriding philosophy for all software work.
- In the modern economy, it is often difficult or impossible to predict how a computer-based system (e.g., a Web-based application) will evolve as time passes.
- Market conditions change rapidly, end-user needs evolve, and new competitive threats emerge without warning.
- In many situations, you won't be able to define requirements fully before the project begins.
- You must be agile enough to respond to a fluid business environment.

What Is Agility?

- An agile team is a quick team able to appropriately respond to changes.
- Agility is an effective response to change.
- Change is what software development is very much about. Changes in the software being built, changes to the team members, changes because of new technology, changes of all kinds that may have an impact on the product they build or the project that creates the product.
  - Agility encourages team structures and attitudes that make communication more facile (too easy).It emphasizes rapid delivery of operational software.
  - It adopts the customer as a part of the development team and recognizes that planning has its limit and that a project plan must be flexible.
  - The time and cost required to ensure that the change is made without unintended side effects is nontrivial.
  - A well-designed agile process flattens the cost of change curve, allowing a software team to accommodate changes late in a software project without dramatic cost and time impact.


Development cost
Cost of change using conventional s/w process
Cost of change using agile process
Idealized cost of change
Development schedule progress

What Is An Agile Process?

- Agile process is characterized in a manner that addresses a number of key assumptions about majority of the software projects:

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

- An agile process must adapt incrementally.

Agile Principles

1. Our first priority is to satisfy the customer through early and continuous delivery of valuable software .
2. Welcome changing requirements, even late in development. Agile processes harness (bind) change for the customer's competitive advantage. This is where it differs from conventional approach.
3. Deliver working software frequently (weeks rather than months)
4. Business people and developers must work together daily throughout the project.
5. Build Projects around motivated individuals. Give them 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 daily cooperation between businesspeople and developers.
7. Working software is the primary measure of progress.

8. Agile process 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 the 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 behavior accordingly.

Human Factors

- Agile development focuses on the talents and skills of individuals, molding the process to specific people and teams.
- Key traits must exist among the people on effective agile software team:

1. **Competence (Skill)** : Competence encompasses innate(inborn) talent, specific software-related skills, and overall knowledge of the process that team has chosen to apply. Skill and knowledge of process can and should be taught to all people who serve as agile team members.
2. **Common focus** : Although members of agile team may perform different tasks and bring different skills to the project, all should be focused on one goal- to deliver a working software increment to the customer within the time promised.
3. **Collaboration** : SE is about assessing, analyzing, and using information that is communicated to the software team; creating information that will help all stakeholders understand the work of the team. To accomplish these tasks, team members must collaborate.

4. **Decision-making ability** : Any good team must be allowed the freedom to control its own destiny. Decision-making authority for both technical and project issues.
5. **Fuzzy problem-solving ability** : Software managers must recognize that the agile team will continually have to deal with ambiguity and will continually be buffeted by change The team must accept the fact that the problem that they are solving today may not be solved tomorrow. Lessons learned from any problem-solving activity may be benefit to the team later in project.
6. **Mutual trust and respect** : The agile team must become a "jelled" team. A jelled team exhibits the trust and respect that are necessary to make them "so strongly knit that the whole is greater than the sum of parts."

7. **Self-organization**: Self-organization implies three things:
   (1) the agile team organizes itself for the work to be done,
   (2) the team organizes the process to best accommodate its local environment,
   (3) the team organizes the work schedule to best achieve delivery of the software increment. (Who will do, what and when).

- Self-organization has a number of technical benefits, but more importantly, it serves to improve collaboration and boost team morale.
- In essence, the team serves as its own management.
- Ken Schwaber addresses these issues when he writes: "The team selects how much work it believes it can perform within the iteration, and the team commits to the work.
- Nothing demotivates a team as much as someone else making commitments for it. Nothing motivates a team as much as accepting the responsibility for fulfilling commitments that it made itself.

Extreme Programming (XP)

- The most widely used approach to agile development, originally proposed by Kent Beck.
- Beck defines a set of five values that establish a foundation for all work performed as part of XP—communication, simplicity, feedback, courage, and respect.
- Each of these values is used as a driver for specific XP activities, actions, and tasks.
- Communication
  - In order to achieve effective *communication between software engineers and* other stakeholders XP emphasizes on informal collaboration, metaphors (Images, smiles, description) , feedback, avoidance of voluminous documentation.
- Simplicity
  - XP restricts developers to design only for immediate needs, rather than consider future needs. The intent is to create a simple design that can be easily implemented in code. If the design must be improved, it can be *refactored at* a later time.

- Feedback
  - Is derived from implemented s/w, customer, other team members.
- Courage
  - might be discipline, s/w team arguing “design for tomorrow” but agile team must have discipline to design for today.
- Respect
  - among team members, stakeholders, indirectly for the s/w itself

XP-Process

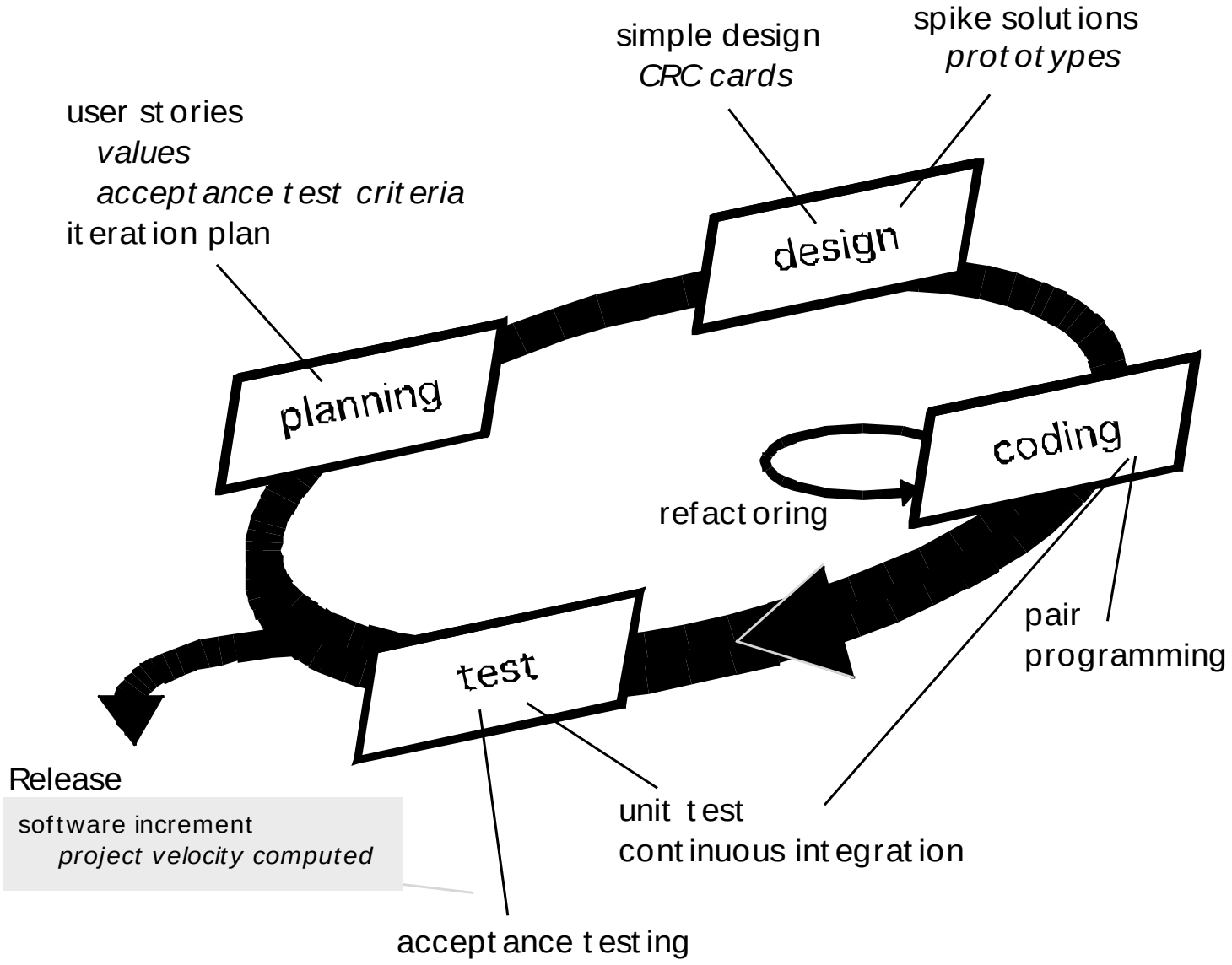
- Extreme Programming uses an object-oriented approach.
- It encompasses a set of rules and practices that occur within the context of four framework activities: planning, design, coding, and testing.
- XP Planning
  - Also called *the planning game,* begins with *listening—a requirements gathering activity 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 and major features and functionality.
  - Listening leads to the creation of "user stories"
  - The customer assigns value to each story that is called as priority
  - Agile team assesses each story and assigns a cost measured in development weeks
  - Customers and developers work together to how to group stories for a deliverable increment

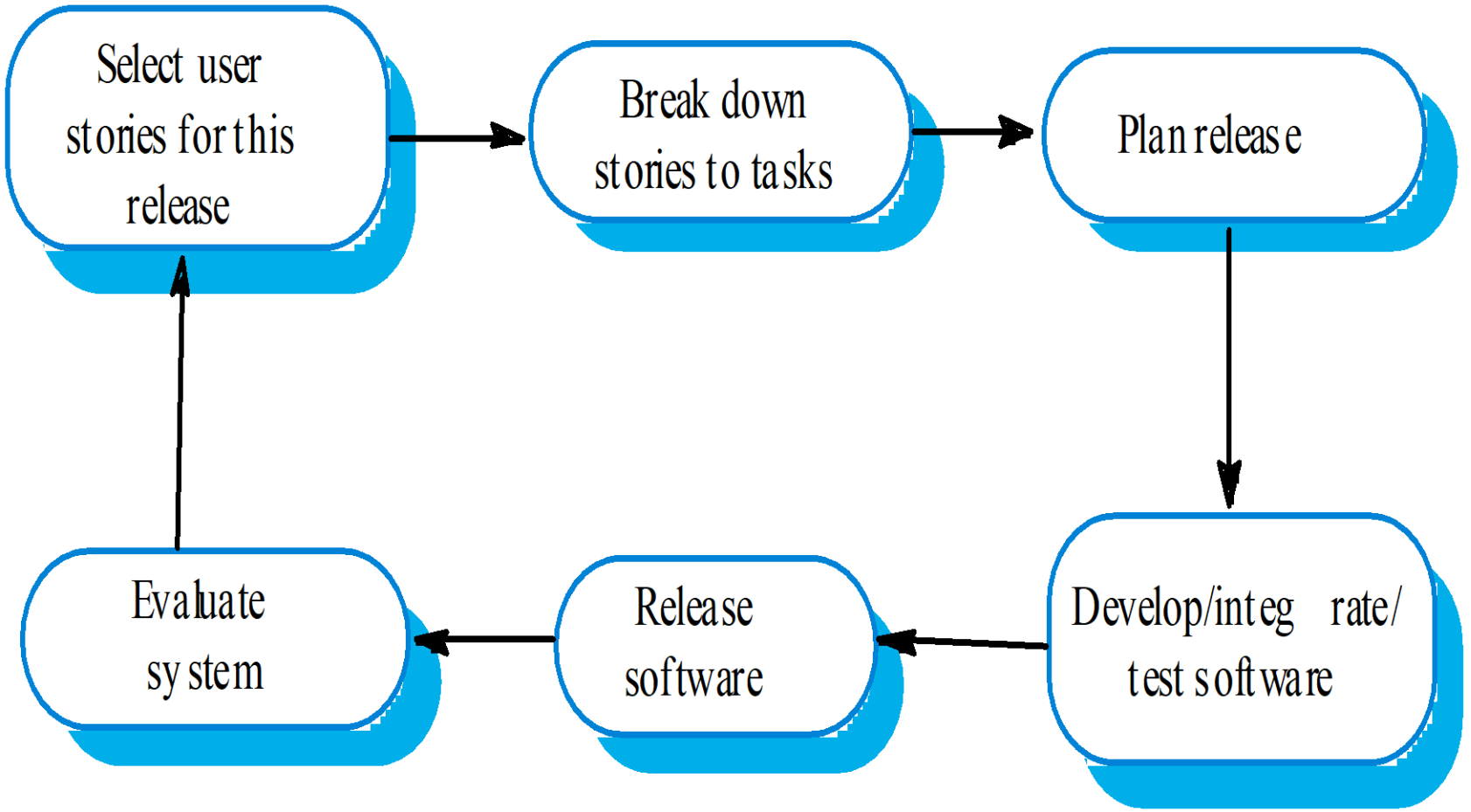
- A commitment is made on delivery date
- The XP team orders the stories that will be developed in one of three ways:
  1. all stories will be implemented immediately (within a few weeks),
  2. the stories with highest value will be moved up in the schedule and implemented first, or
  3. the riskiest stories will be moved up in the schedule and implemented first.
- After the first increment "project velocity" (the number of customer stories implemented during the first release)is computed.
- Project velocity can then be used to (1) help estimate delivery dates and schedule for subsequent releases and (2) determine whether an over commitment has been made for all stories across the entire development project.
- As development work proceeds, the customer can add stories, change the value of an existing story, split stories, or eliminate them.
- The XP team then reconsiders all remaining releases and modifies its plans accordingly.

- XP Design
  - Follows the " Keep It Simple" principle
  - Encourage the use of CRC (class responsibility collaborator) cards (will see in further Chapter) identify and organize OO classes.
  - For difficult design problems, suggests the creation of "spike solutions"—a design prototype to lower risks and to validate original estimates.
  - Encourages "refactoring" (construction technique)-an iterative refinement (modify/simplify) of the internal program design

- XP Coding
  - Recommends the construction of a unit test for a story before coding commences.
  - Once the unit test has been created, the developer is better able to focus on what must be implemented to pass the test.
  - Once the code is complete, it can be unit-tested immediately, thereby providing instantaneous feedback to the developers
  - XP Encourages "pair programming". It recommends that two people work together at one computer workstation to create code for a story.
  - This provides a mechanism for realtime problem solving (two heads are often better than one) and real-time quality assurance (the code is reviewed as it is created).
  - One person might think about the coding details of a particular portion of the design while the other ensures that coding standards are being followed or that the code for the story will satisfy the unit test that has been developed to validate the code against the story.
  - As pair programmers complete their work, the code they develop is integrated with the work of others.

- XP Testing
  - The creation of unit tests before coding commences is a key element of the XP approach
  - All integration and validation testing are executed daily basis (regression testing)
  - It is said that "Fixing small problems every few hours takes less time than fixing huge problems just before the deadline."
  - "Acceptance tests" also called *customer tests,* are defined by the customer and executed to assess customer visible functionality
  - Acceptance tests are derived from user stories that have been implemented as part of a software release.


simple design
CRC cards
spike solutions
prototypes
user stories
values
acceptance test criteria
iteration plan
design
planning
coding
refactoring
test
pair
programming
Release
software increment
project velocity computed
unit test
continuous integration
acceptance testing


Select user stories for this release
Break down stories to tasks
Plan release
Develop/integ rate/ test software
Release software
Evaluate sy stem

Extreme programming practices

Incremental planning	Requirements are recorded on Story Cards and the Stories to be included in a release are determined by the time available and their relative priority. The developers break these Stories into development 'Tasks'.
Small Releases	The minimal useful set of functionality that provides business value is developed first. Releases of the system are frequent and incrementally add functionality to the first release.
Simple Design	Enough design is carried out to meet the current requirements and no more.
Test first development	An automated unit test framework is used to write tests for a new piece of functionality before that functionality itself is implemented.
Refactoring	All developers are expected to refactor the code continuously as soon as possible code improvements are found. This keeps the code simple and maintainable.

Pair Programming	Developers work in pairs, checking each other's work and providing the support to always do a good job.
Collective Ownership	The pairs of developers work on all areas of the system, so that no islands of expertise develop and all the developers own all the code. Anyone can change anything.
Continuous Integration	As soon as work on a task is complete it is integrated into the whole system. After any such integration, all the unit tests in the system must pass.
Sustainable pace	Large amounts of over-time are not considered acceptable as the net effect is often to reduce code quality and medium term productivity
On-site Customer	A representative of the end-user of the system (the Customer) should be available full time for the use of the XP team. In an extreme programming process, the customer is a member of the development team and is responsible for bringing system requirements to the team for implementation.

XP Debate

- **What are some of the issues that lead to an XP debate?**

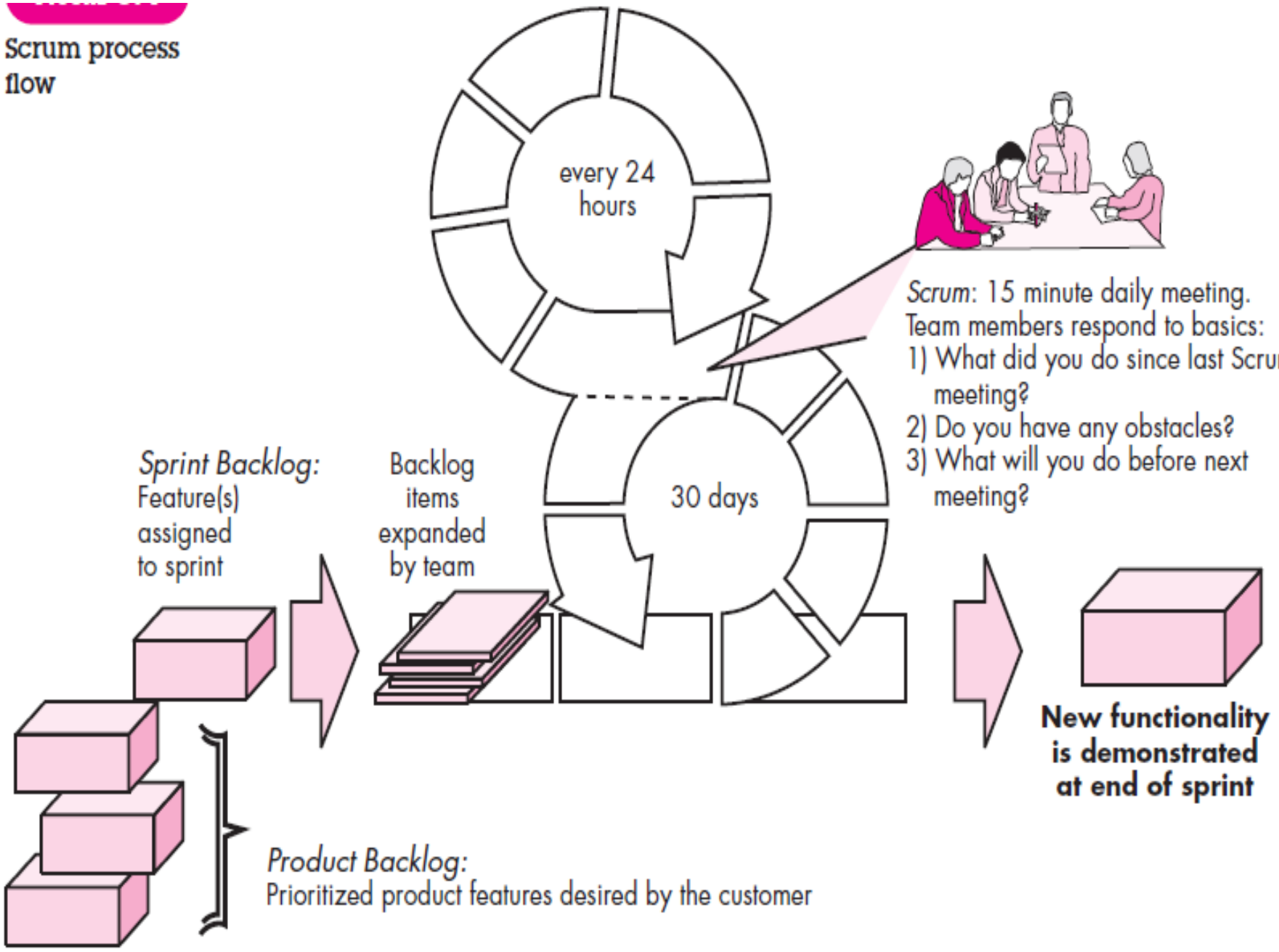
1. *Requirements volatility*. Because the customer is an active member of the *XP* team, changes to requirements are requested informally. As a consequence, the scope of the project can change and earlier work may have to be modified to accommodate current needs.
2. *Conflicting customer needs*. Many projects have multiple customers, each *with* his own set of needs
3. *Requirements are expressed informally*. User stories and acceptance tests are the only explicit manifestation of requirements in XP. Critics argue that a more formal model or specification is often needed to ensure that omissions, inconsistencies, and errors are uncovered before the system is built.
4. *Lack of formal design*. XP deemphasizes the need for architectural design and in many instances, suggests that design of all kinds should be relatively informal. Critics argue that when complex systems are built, design must be emphasized to ensure that the overall structure of the software will exhibit quality and maintainability.

- Every software process has flaws and that many software organizations have used XP successfully. The key is to recognize where a process may have weaknesses and to adapt it to the specific needs of your organization.

SCRUM

- Scrum (the name is derived from an activity that occurs during a rugby match) is an agile software development method that was conceived by Jeff Sutherland and his development team in the early 1990s. In recent years, further development on the Scrum methods has been performed by Schwaber and Beedle.
- Scrum principles are consistent with the agile manifesto and are used to guide development activities within a process that incorporates the following framework activities: requirements, analysis, design, evolution, and delivery.
- Within each framework activity, work tasks occur within a process pattern called a *sprint*. The work conducted within a sprint (the number of sprints required for each framework activity will vary depending on product complexity and size) is adapted to the problem at hand and is defined and often modified in real time by the Scrum team. The overall flow of the Scrum process is illustrated in figure:

Scrum process flow



- Scrum emphasizes the use of a set of software process patterns that have proven effective for projects with tight timelines, changing requirements, and business criticality. Each of these process patterns defines a set of development actions:
- *Backlog—a prioritized list of project requirements or features that provide business* value for the customer. Items can be added to the backlog at any time (this is how changes are introduced). The product manager assesses the backlog and updates priorities as required.
- *Sprints—consist of work units that are required to achieve a requirement defined* in the backlog that must be fit into a predefined time-box Changes (e.g., backlog work items) are not introduced during the sprint. Hence, the sprint allows team members to work in a short-term, but stable environment.

- *Scrum meetings—are short (typically 15 minutes) meetings held daily by the Scrum* team. Three key questions are asked and answered by all team members:
  - What did you do since the last team meeting?
  - What obstacles are you encountering?
  - What do you plan to accomplish by the next team meeting?
- A team leader, called a *Scrum master, leads the meeting and assesses the responses* from each person. The Scrum meeting helps the team to uncover potential problems as early as possible. Also, these daily meetings lead to "knowledge socialization" and thereby promote a self-organizing team structure.
- *Demos—deliver the software increment to the customer so that functionality that* has been implemented can be demonstrated and evaluated by the customer. It is important to note that the demo may not contain all planned functionality, but rather those functions that can be delivered within the time-box that was established.

Chapter : 4

Principles that Guide Practice

SE practice

- **What is software engineering "practice"?**
  - Practice is a collection of concepts, principles, methods, and tools that a software engineer calls upon on a daily basis.
  - Practice allows managers to manage software projects and software engineers to build computer programs.
  - Practice populates a software process model with the necessary technical and management how-to's to get the job done.
  - Practice transforms a haphazard unfocused approach into something that is more organized, more effective, and more likely to achieve success.

- You often hear people say that software development knowledge has a 3-year half-life: half of what you need to know today will be obsolete within 3 years. (Concept comes from any radio active material having a half life).
- In the domain of technology-related knowledge (C, C++, JAVA, etc) that's probably about right. But there is another kind of software development knowledge—a kind that I think of as "software engineering principles"—that does not have a three-year half-life.
- These software engineering principles are likely to serve a professional programmer throughout his or her career.

CORE PRINCIPLES

- Software engineering is guided by a collection of core principles that help in the application of a meaningful software process and the execution of effective software engineering methods.
- At the process level, core principles establish a philosophical foundation that guides a software team as it performs framework and umbrella activities, navigates the process flow, and produces a set of software engineering work products.
- At the level of practice, core principles establish a collection of values and rules that serve as a guide as you analyze a problem, design a solution, implement and test the solution, and ultimately deploy the software in the user community.

Principles That Guide Process

- **Principle 1. *Be agile.***
  - Whether the process model you choose is prescriptive or agile, the basic tenets of agile development should govern your approach. Every aspect of the work you do should emphasize economy of action—keep your technical approach as simple as possible, keep the work products you produce as concise as possible, and make decisions locally whenever possible.
- **Principle 2.**
  - Focus on quality at every step. The exit condition for every process activity, action, and task should focus on the quality of the work product that has been produced.
- **Principle 3.**
  - Be ready to adapt. Process is not a religious experience, and dogma has no place in it. When necessary, adapt your approach to constraints imposed by the problem, the people, and the project itself.

- **Principle 4.** ***Build an effective team.***
  - Software engineering process and practice are important, but the bottom line is people. Build a self-organizing team that has mutual trust and respect.
- **Principle 5.** ***Establish mechanisms for communication and coordination.***
  - Projects fail because important information falls into the cracks and/or stakeholders fail to coordinate their efforts to create a successful end product. These are management issues and they must be addressed.
- **Principle 6.** ***Manage change.***
  - The approach may be either formal or informal, but mechanisms must be established to manage the way changes are requested, assessed, approved, and implemented.

- **Principle 7.** ***Assess risk.***
  - Lots of things can go wrong as software is being developed. It's essential that you establish contingency plans.
- **Principle 8.** ***Create work products that provide value for others.***
  - Create only those work products that provide value for other process activities, actions, or tasks. Every work product that is produced as part of software engineering practice will be passed on to someone else. A list of required functions and features will be passed along to the person (people) who will develop a design, the design will be passed along to those who generate code, and so on. Be sure that the work product imparts the necessary information without ambiguity or omission.

Principles That Guide Practice

- Software engineering practice has a single overriding goal—to deliver on-time, highquality, operational software that contains functions and features that meet the needs of all stakeholders.
- To achieve this goal, you should adopt a set of core principles that guide your technical work.
- **Principle 1.** ***Divide and conquer.***
  - Stated in a more technical manner, analysis and design should always emphasize separation of concerns (SoC). A large problem is easier to solve if it is subdivided into a collection of elements (or concerns). Ideally, each concern delivers distinct functionality that can be developed, and in some cases validated, independently of other concerns.

- **Principle 2.** ***Understand the use of abstraction.***
  - At its core, an abstraction is a simplification of some complex element of a system used to communicate meaning in a single phrase.
  - In software engineering practice, you use many different level of abstraction, each imparting or implying meaning that must be communicated.
  - In analysis and design work, a software team normally begins with models that represent high levels of abstraction (e.g., a spreadsheet) and slowly refines those models into lower levels of abstraction (e.g., a column or the SUM function).
- **Principle 3. Strive for consistency.**
  - Whether it's creating a requirements model, developing a software design, generating source code, or creating test cases, the principle of consistency suggests that a familiar context makes software easier to use.
  - As an example, consider the design of a user interface for a WebApp. Consistent placement of menu options, the use of a consistent color scheme, and the consistent use of recognizable icons all help to make the interface ergonomically sound.

- **Principle 4. Focus on the transfer of information.**
  - Software is about information transfer—from a database to an end user, from a legacy system to a WebApp, from an end user into a graphic user interface (GUI), from an operating system to an application, from one software component to another— the list is almost endless.
  - In every case, information flows across an interface, and as a consequence, there are opportunities for error, or omission, or ambiguity.
  - The implication of this principle is that you must pay special attention to the analysis, design, construction, and testing of interfaces.
- **Principle 5. Build software that exhibits effective modularity.**
  - Separation of concerns (Principle 1) establishes a philosophy for software.
  - Modularity provides a mechanism for realizing the philosophy.
  - Any complex system can be divided into modules (components), but good software engineering practice demands more.
  - Modularity must be effective. That is, each module should focus exclusively on one well-constrained aspect of the system—it should be cohesive in its function and/or constrained in the content it represents.
  - Additionally, modules should be interconnected in a relatively simple manner—each module should exhibit low coupling to other modules, to data sources, and to other environmental aspects.

- **Principle 6.** ***Look for patterns.***
  - The goal of patterns within the software community is to create a body of literature to help software developers resolve recurring problems encountered throughout all of software development.
  - Patterns help create a shared language for communicating insight and experience about these problems and their solutions.
  - Formally codifying these solutions and their relationships lets us successfully capture the body of knowledge which defines our understanding of good architectures that meet the needs of their users.
- **Principle 7.** ***When possible, represent the problem and its solution from a number of different perspectives.***
  - When a problem and its solution are examined from a number of different perspectives, it is more likely that greater insight will be achieved and that errors and omissions will be uncovered.
  - For example, a requirements model can be represented using a data oriented viewpoint, a function-oriented viewpoint, or a behavioral viewpoint. Each provides a different view of the problem and its requirements.

- **Principle 8.** ***Remember that someone will maintain the software.***
- ***Over*** the long term, software will be corrected as defects are uncovered, adapted as its environment changes, and enhanced as stakeholders request more capabilities.
- These maintenance activities can be facilitated if solid software engineering practice is applied throughout the software process.

Communication Principles

- Before customer requirements can be analyzed, modeled, or specified they must be gathered through the communication activity.
- Effective communication (among technical peers, with the customer and other stakeholders, and with project managers) is among the most challenging activities that you will confront.
- **Principle 1. *Listen*.**
  - Try to focus on the speaker's words, rather than formulating your response to those words. Ask for clarification if something is unclear, but avoid constant interruptions.
  - Never become contentious in your words or actions (e.g., rolling your eyes or shaking your head) as a person is talking.
- **Principle 2. *Prepare before you communicate*.**
  - Spend the time to understand the problem before you meet with others.
  - If necessary, do some research to understand business domain jargon.
  - If you have responsibility for conducting a meeting, prepare an agenda in advance of the meeting.

- **Principle 3.** ***Someone should facilitate the activity.***
  - Every communication meeting should have a leader (a facilitator) to keep the conversation moving in a productive direction, (2) to mediate any conflict that does occur, and (3) to ensure than other principles are followed.
- **Principle 4.** ***Face-to-face communication is best.***
  - But it usually works better when some other representation of the relevant information is present. For example, a participant may create a drawing or a "strawman" document that serves as a focus for discussion.
- **Principle 5.** ***Take notes and document decisions.***
  - Things have a way of falling into the cracks. Someone participating in the communication should serve as a "recorder" and write down all important points and decisions.

- **Principle 6.** ***Strive for collaboration.***
  - Collaboration and consensus occur when the collective knowledge of members of the team is used to describe product or system functions or features.
  - Each small collaboration serves to build trust among team members and creates a common goal for the team.
- **Principle 7.** ***Stay focused; modularize your discussion.***
  - The more people involved in any communication, the more likely that discussion will bounce from one topic to the next.
  - The facilitator should keep the conversation modular, leaving one topic only after it has been resolved.
- **Principle 8.** ***If something is unclear, draw a picture.***
  - Verbal communication goes only so far. A sketch or drawing can often provide clarity when words fail to do the job.

- **Principle 9.** ***(a) Once you agree to something, move on. (b) If you can't agree to something, move on. (c) If a feature or function is unclear and cannot be clarified at the moment, move on.***
  - Communication, like any software engineering activity, takes time. Rather than iterating endlessly, the people who participate should recognize that many topics require discussion (see Principle 2) and that "moving on" is sometimes the best way to achieve communication agility.
- **Principle 10.** ***Negotiation is not a contest or a game. It works best when both parties win.***
  - There are many instances in which you and other stakeholders must negotiate functions and features, priorities, and delivery dates. If the team has collaborated well, all parties have a common goal. Still, negotiation will demand compromise from all parties.

Planning Principles

- The communication activity helps you to define your overall goals and objectives (subject, of course, to change as time passes).
- However, understanding these goals and objectives is not the same as defining a plan for getting there.
- The planning activity encompasses a set of management and technical practices that enable the software team to define a road map as it travels toward its strategic goal and tactical objectives.
- Try as we might, it's impossible to predict exactly how a software project will evolve.
- There is no easy way to determine what unforeseen technical problems will be encountered, what important information will remain undiscovered until late in the project, what misunderstandings will occur, or what business issues will change.
- And yet, a good software team must plan its approach.

- **Principle 1.** ***Understand the scope of the project.***
  - It's impossible to use a road map if you don't know where you're going. Scope provides the software team with a destination.
- **Principle 2.** ***Involve stakeholders in the planning activity.***
  - Stakeholders define priorities and establish project constraints.
  - To accommodate these realities, software engineers must often negotiate order of delivery, time lines, and other project-related issues.
- **Principle 3.** ***Recognize that planning is iterative.***
  - A project plan is never engraved in stone. As work begins, it is very likely that things will change.
  - As a consequence, the plan must be adjusted to accommodate these changes.
  - In addition, iterative, incremental process models dictate replanning after the delivery of each software increment based on feedback received from users.

- **Principle 4.** ***Estimate based on what you know.***
  - The intent of estimation is to provide an indication of effort, cost, and task duration, based on the team's current understanding of the work to be done.
  - If information is vague or unreliable, estimates will be equally unreliable.
- **Principle 5.** ***Consider risk as you define the plan.***
  - If you have identified risks that have high impact and high probability, contingency planning is necessary.
  - In addition, the project plan (including the schedule) should be adjusted to accommodate the likelihood that one or more of these risks will occur.
- **Principle 6.** ***Be realistic.***
  - People don't work 100 percent of every day. Noise always enters into any human communication.
  - Omissions and ambiguity are facts of life. Change will occur. Even the best software engineers make mistakes.
  - These and other realities should be considered as a project plan is established.

- **Principle 7.** ***Adjust granularity as you define the plan.***
  - Granularity refers to the level of detail that is introduced as a project plan is developed.
  - A "high-granularity" plan provides significant work task detail that is planned over relatively short time increments (so that tracking and control occur frequently).
  - A "low-granularity" plan provides broader work tasks that are planned over longer time periods. In general, granularity moves from high to low as the project time line moves away from the current date.
  - Over the next few weeks or months, the project can be planned in significant detail.
  - Activities that won't occur for many months do not require high granularity (too much can change).
- **Principle 8.** ***Define how you intend to ensure quality.***
  - The plan should identify how the software team intends to ensure quality.
  - If technical reviews3 are to be conducted, they should be scheduled. If pair programming is to be used during construction, it should be explicitly defined within the plan.

- **Principle 9.** ***Describe how you intend to accommodate change.***
  - Even the best planning can be obviated by uncontrolled change. You should identify how changes are to be accommodated as software engineering work proceeds.
  - For example, can the customer request a change at any time? If a change is requested, is the team obliged to implement it immediately? How is the impact and cost of the change assessed?
- **Principle 10.** ***Track the plan frequently and make adjustments as required.***
  - Software projects fall behind schedule one day at a time.
  - Therefore, it makes sense to track progress on a daily basis, looking for problem areas and situations in which scheduled work does not conform to actual work conducted.
  - When slippage is encountered, the plan is adjusted accordingly.

Modeling Principles

- We create models to gain a better understanding of the actual entity to be built.
- When the entity is a physical thing (e.g., a building, a plane, a machine), we can build a model that is identical in form and shape but smaller in scale.
- However, when the entity to be built is software, our model must take a different form. It must be capable of representing the information that software transforms, the architecture and functions that enable the transformation to occur, the features that users desire, and the behavior of the system as the transformation is taking place.
- Models must accomplish these objectives at different levels of abstraction—first depicting the software from the customer's viewpoint and later representing the software at a more technical level.
- In software engineering work, two classes of models can be created:
  - Requirements models and
  - design models.
- Requirements models (also called analysis models) represent customer requirements by depicting the software in three different domains: the information domain, the functional domain, and the behavioral domain.
- Design models represent characteristics of the software that help practitioners to construct it effectively: the architecture, the user interface, and component-level detail.

- **Principle 1.** ***The primary goal of the software team is to build software, not create models.***
  - Agility means getting software to the customer in the fastest possible time. Models that make this happen are worth creating, but models that slow the process down or provide little new insight should be avoided.
- **Principle 2.** ***Travel light—don't create more models than you need.***
  - Every model that is created must be kept up-to-date as changes occur. More importantly, every new model takes time that might otherwise be spent on construction (coding and testing). Therefore, create only those models that make it easier and faster to construct the software.
- **Principle 3.** ***Strive to produce the simplest model that will describe the problem or the software. Don't overbuild the software.***
  - By keeping models simple, the resultant software will also be simple. The result is software that is easier to integrate, easier to test, and easier to maintain (to change). In addition, simple models are easier for members of the software team to understand and critique, resulting in an ongoing form of feedback that optimizes the end result.

- **Principle 4.** ***Build models in a way that makes them amenable to change.***
  - Assume that your models will change, but in making this assumption don't get sloppy. For example, since requirements will change, there is a tendency to give requirements models short shrift. Why? Because you know that they'll change anyway. The problem with this attitude is that without a reasonably complete requirements model, you'll create a design (design model) that will invariably miss important functions and features.
- **Principle 5.** ***Be able to state an explicit purpose for each model that is created.***
  - Every time you create a model, ask yourself why you're doing so. If you can't provide solid justification for the existence of the model, don't spend time on it.
- **Principle 6.** ***Adapt the models you develop to the system at hand.***
  - It may be necessary to adapt model notation or rules to the application; for example, a video game application might require a different modeling technique than real-time, embedded software that controls an automobile engine.

- **Principle 7.** ***Try to build useful models, but forget about building perfect models.***
  - When building requirements and design models, a software engineer reaches a point of diminishing returns. That is, the effort required to make the model absolutely complete and internally consistent is not worth the benefits of these properties. Am I suggesting that modeling should be sloppy or low quality? The answer is "no." But modeling should be conducted with an eye to the next software engineering steps. Iterating endlessly to make a model "perfect" does not serve the need for agility.
- **Principle 8.** ***Don't become dogmatic about the syntax of the model.***
  - If it communicates content successfully, representation is secondary. Although everyone on a software team should try to use consistent notation during modeling, the most important characteristic of the model is to communicate information that enables the next software engineering task. If a model does this successfully, incorrect syntax can be forgiven.

- **Principle 9.** ***If your instincts tell you a model isn't right even though it***
- ***seems okay on paper, you probably have reason to be concerned. If***
- you are an experienced software engineer, trust your instincts. Software
- work teaches many lessons—some of them on a subconscious level. If something
- tells you that a design model is doomed to fail (even though you can't
- prove it explicitly), you have reason to spend additional time examining the
- model or developing a different one.
- **Principle 10.** ***Get feedback as soon as you can. Every model should be***
- reviewed by members of the software team. The intent of these reviews is to
- provide feedback that can be used to correct modeling mistakes, change misinterpretations,
- and add features or functions that were inadvertently omitted.

Analysis Modeling Principles

- There are 5 such principles.
- **Principle 1.** ***The information domain of a problem must be represented and understood.***
  - ***The information domain encompasses the data that flow*** into the system (from end users, other systems, or external devices), the data that flow out of the system (via the user interface, network interfaces, reports, graphics, and other means), and the data stores that collect and organize persistent data objects (i.e., data that are maintained permanently).
- **Principle 2.** ***The functions that the software performs must be defined.***
  - Software functions provide direct benefit to end users and also provide internal support for those features that are user visible.
  - Some functions transform data that flow into the system. In other cases, functions effect some level of control over internal software processing or external system elements. Functions can be described at many different levels of abstraction, ranging from a general statement of purpose to a detailed description of the processing elements that must be invoked.

- **Principle 3.** ***The behavior of the software (as a consequence of external***
- ***events) must be represented.***
  - The behavior of computer software is driven by its interaction with the external environment. Input provided by end users, control data provided by an external system, or monitoring data collected over a network all cause the software to behave in a specific way.
- **Principle 4.** ***The models that depict information, function, and behavior must be partitioned in a manner that uncovers detail in a layered (or hierarchical) fashion.***
  - Requirements modeling is the first step in software engineering problem solving. It allows you to better understand the problem and establishes a basis for the solution (design).
  - Complex problems are difficult to solve in their entirety. For this reason, you should use a divide-and-conquer strategy. A large, complex problem is divided into subproblems until each subproblem is relatively easy to understand. This concept is called partitioning or separation of concerns, and it is a key strategy in requirements modeling.

- **Principle 5.** ***The analysis task should move from essential information toward implementation detail.***
  - Requirements modeling begins by describing the problem from the end-user's perspective. The "essence" of the problem is described without any consideration of how a solution will be implemented. For example, a video game requires that the player "instruct" its protagonist on what direction to proceed as she moves into a dangerous maze.
  - That is the essence of the problem. Implementation detail (normally described as part of the design model) indicates how the essence will be implemented. For the video game, voice input might be used.
  - Alternatively, a keyboard command might be typed, a joystick (or mouse) might be pointed in a specific direction, or a motion-sensitive device might be waved in the air.

Design Modeling Principles

- The software design model is like the architect's plan for the house.
- It begins by representing the totality of the thing to be built i.e three dimensional rendering of the house and then refines the thing to provide guidance for constructing each detail (plumbing, electrical wiring, air conditioning ducting etc).
- Similarly the design model that is created for software provides a variety of different views of the system.
- There are 9 principles that are useful while doing design modeling. They are as follows :

- **Principle 1.** ***Design should be traceable to the requirements model.***
  - The requirements model describes the information domain of the problem, user-visible functions, system behavior, and a set of requirements classes that package business objects with the methods that service them. The design model translates this information into an architecture, a set of subsystems that implement major functions, and a set of components that are the realization of requirements classes. The elements of the design model should be traceable to the requirements model.
- **Principle 2.** ***Always consider the architecture of the system to be built.***
  - Software architecture is the skeleton of the system to be built. It affects interfaces, data structures, program control flow and behavior, the manner in which testing can be conducted, the maintainability of the resultant system, and much more. For all of these reasons, design should start with architectural considerations. Only after the architecture has been established should component-level issues be considered.
- **Principle 3.** ***Design of data is as important as design of processing functions.***
  - Data design is an essential element of architectural design. The manner in which data objects are realized within the design cannot be left to chance. A well-structured data design helps to simplify program flow, makes the design and implementation of software components easier, and makes overall processing more efficient.

- **Principle 4.** ***Interfaces (both internal and external) must be designed with care.***
  - The manner in which data flows between the components of a system has much to do with processing efficiency, error propagation, and design simplicity. A well-designed interface makes integration easier and assists the tester in validating component functions.
- **Principle 5.** ***User interface design should be tuned to the needs of the end user.***
  - However, in every case, it should stress ease of use. The user interface is the visible manifestation of the software. No matter how sophisticated its internal functions, no matter how comprehensive its data structures, no matter how well designed its architecture, a poor interface design often leads to the perception that the software is "bad."
- **Principle 6.** ***Component-level design should be functionally independent.***
  - Functional independence is a measure of the "single-mindedness" of a software component. The functionality that is delivered by a component should be cohesive—that is, it should focus on one and only one function or subfunction.

- **Principle 7.** ***Components should be loosely coupled to one another and to the external environment.***
  - Coupling is achieved in many ways—via a component interface, by messaging, through global data. As the level of coupling increases, the likelihood of error propagation also increases and the overall maintainability of the software decreases. Therefore, component coupling should be kept as low as is reasonable.
- **Principle 8.** ***Design representations (models) should be easily understandable.***
  - The purpose of design is to communicate information to practitioners who will generate code, to those who will test the software, and to others who may maintain the software in the future. If the design is difficult to understand, it will not serve as an effective communication medium.
- **Principle 9.** ***The design should be developed iteratively.***
  - With each iteration, the designer should strive for greater simplicity. Like almost all creative activities, design occurs iteratively. The first iterations work to refine the design and correct errors, but later iterations should strive to make the design as simple as is possible.

Construction Principles

- The construction activity encompasses a set of coding and testing tasks that lead to operational software that is ready for delivery to the customer or the end-user.
- Here, coding may mean
  - The direct (manual) creation of programming language source code,
  - the automatic generation of source code using an intermediate design-like representation of the component to be built or
  - the automatic generation of executable code using a fourth generation programming language like Visual C++ (where you do drag and drop).
- There are different levels of testing like unit testing of component, integration testing conducted as the system is constructed, validation testing that assesses whether requirements have been met and acceptance testing that is conducted by the customer.
- There are fundamental principles and concepts which are applicable to coding and testing. They are as follows:

Coding Principles and Concepts

- These principles are divided into 3 parts :
  - preparation principles (things you need to understand and do before you start writing code),
  - coding principles (things you understand and do while you are writing the code) and
  - validation principles (things you do after coding is done).
- Preparation principles. Before you write one line of code, be sure you
  - Understand the problem you are trying to solve.
  - Understand basic design concepts and principles.
  - Pick a programming language that meets the needs of the software to be built and the environment in which it will operate.
  - Select programming environment that provides tools that will make your work easier.
  - Create a set of unit tests that will be applied once the component you code is completed.

- Coding Principles. As you begin writing code, be sure you
  - Construct your algorithms by following structured programming practice.
  - Select data structures that will meet the needs of the design.
  - Understand the software architecture and create interfaces that are consistent with it.
  - Keep conditional logic as simple as possible.
  - Create nested loops in a way that makes them easily testable.
  - Select meaningful variable names and follow other local coding standards.
  - Write code that is self documenting.
  - Create a visual layout (indenting, spacing) that aids understanding.
- Validation Principles. After you have completed the first coding pass, be sure you
  - Conduct a code walkthrough when appropriate (peer review)
  - Perform unit tests and correct the errors you have uncovered.
  - Refactor the code. (see if it can be improved further)

- **Testing Principles**
- There are number of rules that can serve well as testing objectives
  - Testing is the process of executing a program with the intent of finding an error.
  - A good test case is one that has a high probability of finding an as-yet undiscovered error.
  - A successful test is one that uncovers an as-yet-undiscovered error.
- The main objective of testing is to find errors. It is exactly the opposite of the view held by most developers that a successful test is one in which no errors are found.
- Our objective is to design tests that systematically uncover different classes of errors and to do so with a minimum amount of time and error.
- One needs to understand following five testing principles.

- **Principle 1: All tests should be traceable to customer requirements**
  - The objective of software testing is to uncover errors. It follows that the most severe defects (from the customer's point of view) are those that cause the program to fail to meet its requirements.
- **Principle 2:Tests should be planned long before testing begins.**
  - Test planning can begin as soon as the requirements model is complete.
  - Detailed definition of test cases can begin as soon as the design model has been solidified. Therefore, all tests can be planned and designed before any code has been generated..
- **Principle 3:The pareto principle applies to software testing.**
  - In this context the Pareto principle implies that 80 percent of all errors uncovered during testing will likely be traceable to 20 percent of all program components.
  - The problem, of course, is to isolate these suspect components and to thoroughly test them.
  - Identify these 20 % components and do thorough testing of these components.

- **Principle 4:Testing should begin "in the small" and progress toward testing "in the large".**
  - Start with individual components and then move to clusters of components (modules) and then to the whole system.
- **Principle 5: Exhaustive (complete, full, comprehensive) testing is not possible.** It takes too much time and effort and the cost may not give enough return on the investment.

Deployment Principles

- Once the software has been tested properly it has to be deployed. The deployment activity consists of three actions : delivery, support and feedback.
- Because modern software process models are evolutionary or incremental in nature, deployment happens not once, but a number of times as software moves toward completion.
- Each delivery cycle provides the customer and end users with an operational software increment that provides usable functions and features.
- Each support cycle provides documentation and human assistance for all functions and features introduced during all deployment cycles to date.
- Each feedback cycle provides the software team with important guidance that results in modifications to the functions, features, and approach taken for the next increment.
- The delivery of a software increment represents an important milestone for any software project.
- A number of key principles should be followed as the team prepares to deliver an increment:

- There are following 5 principles that guide in this activity.
  1. **Customer expectations for the software must be managed.**

     Do not raise the expectations too high by promising too much if you are not going to deliver it.
  2. **A complete delivery package should be assembled and tested.**

     A CD-ROM or other media (including Web-based downloads) containing all executable software, support data files, support documents, and other relevant information should be assembled and thoroughly beta-tested with actual users.

     All installation scripts and other operational features should be thoroughly exercised in as many different computing configurations (i.e., hardware, operating systems, peripheral devices, networking arrangements) as possible.

3. **A support regime must be established before the software is delivered.**
   - An end user expects responsiveness and accurate information when a question or problem arises.
   - If support is ad hoc, or worse, nonexistent, the customer will become dissatisfied immediately.
   - Support should be planned, support materials should be prepared, and appropriate recordkeeping mechanisms should be established so that the software team can conduct a categorical assessment of the kinds of support requested.
4. **Appropriate instructional materials must be provided to the end-users.**
   - The software team delivers more than the software itself.
   - Appropriate training aids (if required) should be developed; troubleshooting guidelines should be provided, and when necessary, a "what's different about this software increment" description should be published.

5. **Buggy software should be fixed first, delivered later.**

- Under time pressure, some software organizations deliver low-quality increments with a warning to the customer that bugs "will be fixed in the next release."
- This is a mistake. There's a saying in the software business: "Customers will forget you delivered a high-quality product a few days late, but they will never forget the problems that a low-quality product caused them.
- The software reminds them every day."