# Unit-1 Micro Syllabus

- Introduction to Software Engineering:
- Software- Define Software
- Software Engineering- Fundamentals to Software Engineering and the importance of Software Engineering.
- The nature of software.
- The changing nature of software
- Software Process: Software engineering-A layered technology.
- A process framework-
- 1. Defining a framework Activity 2. Identifying the Task set 3. Process Patterns

- Software engineering practice
- Software development myths-
- Management myths , Customer Myths, Practitioner's myths

## Process models:

- Prescriptive Process Models
  - Waterfall model, Incremental process model, Prototyping and spiral models
- Unified Process Model.
- Agile process model: SCRUM

# The Nature of Software

Software takes on a dual role.

- It is a product,
- The vehicle for delivering a product.

**As a product**,

- ❖ it delivers the computing potential embodied by computer hardware, by a network of computers that are accessible by local hardware.
- ❖ Whether it resides within a mobile phone or operates inside a mainframe computer,
- ❖ software is an information transformer—**producing, managing, acquiring, modifying, displaying, or transmitting information** that can be as simple as a single bit or as complex as a multimedia presentation derived from data acquired from dozens of independent sources.

# Contd…

Software delivers the most important product of our time—*information.*

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

# Contd…

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.

# Contd…

The questions that were asked of the lone programmer are the same questions that are asked when modern computer-based systems are built:

- **Why does it take so long to get software finished?**
- **Why are development costs so high?**
- **Why can't we find all errors before we give the software to our customers?**
- **Why do we spend so much time and effort maintaining existing programs?**
- **Why do we continue to have difficulty in measuring progress as software is being developed and maintained?**

# Definition of Software

*Software is:*

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

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

*(3) documentation that describes the operation and use of the programs.*

# Characteristics of Software:

- Software is a **logical rather than a physical system element**. Therefore, software has characteristics that are considerably different than those of hardware:

1. ***Software is developed or engineered, it is not manufactured in the classical sense.***
2. ***Software doesn't "wear out."***
3. ***Although the industry is moving toward component-based construction, most software continues to be custom-built.***

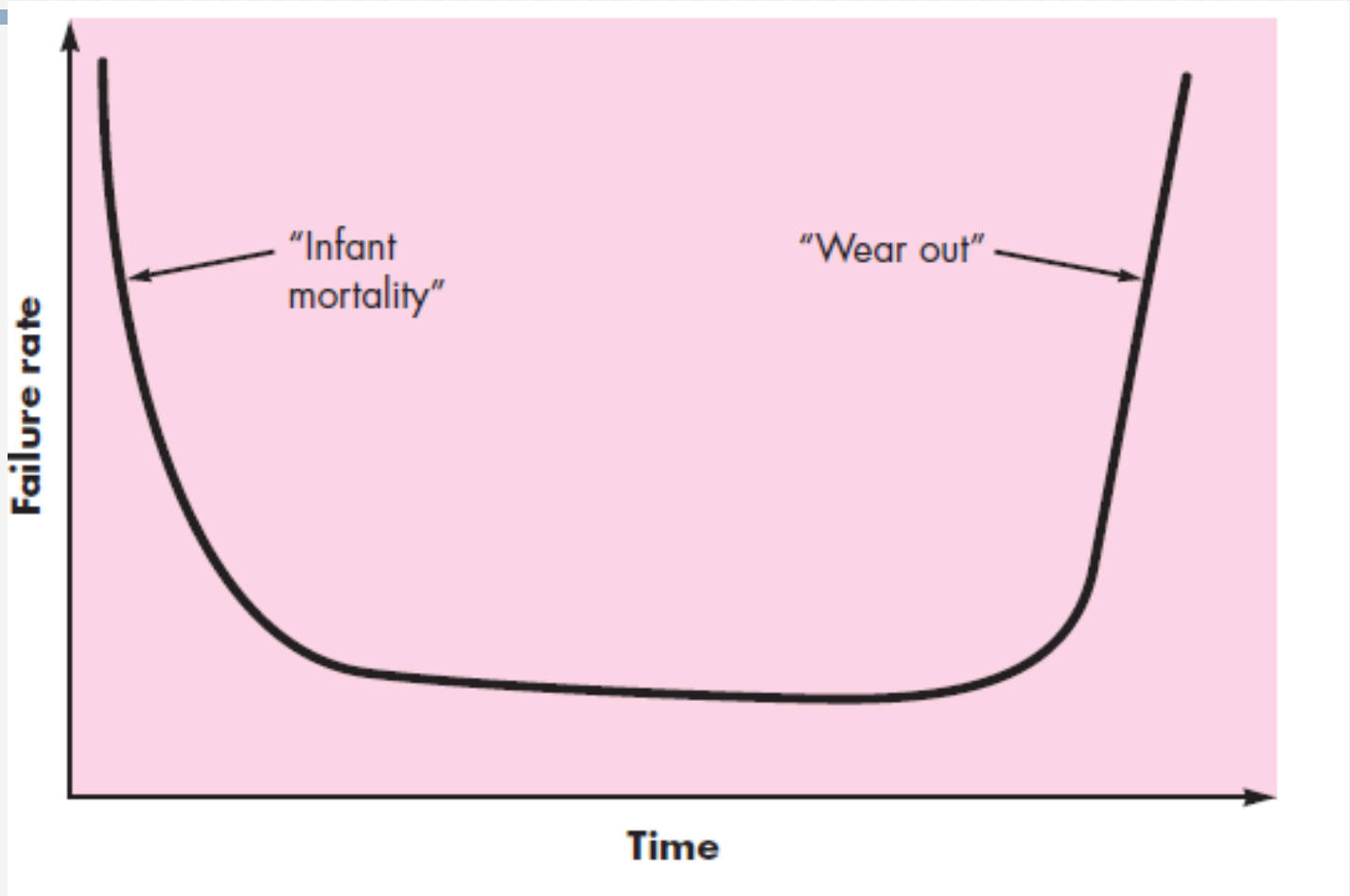
# Characteristics of Software

- ***Software is developed or engineered, it is not manufactured in the classical sense.***
- ***So***ftware development and hardware manufacturing, the two activities are fundamentally different.
- In both activities, high quality is achieved through good design, but the **manufacturing phase for hardware can introduce quality problems** that are non-existent(or easily corrected) for software.
- Both activities are dependent on people, but the relationship between people applied and work accomplished is entirely different.
- Both activities require the construction of a “product,” but the approaches are different.

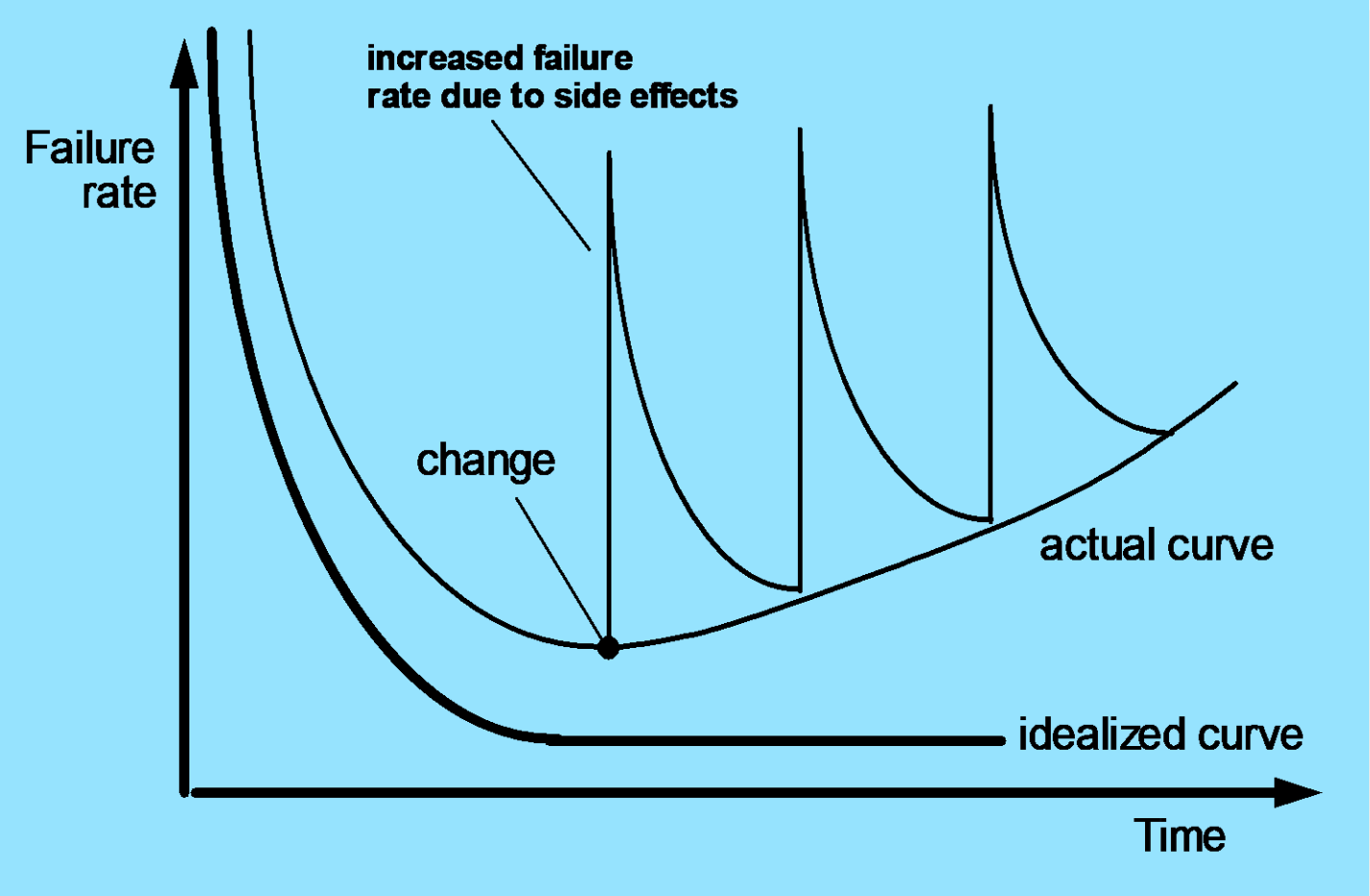
# Characteristics of Software?

- ***Software doesn't "wear out."***
- Figure 1.1 depicts failure rate as a function of time for hardware.
- The relationship, often called the “bathtub curve,” indicates that hardware exhibits relatively high failure rates early in its life (these failures are often attributable to design or manufacturing defects);
- defects are corrected and the failure rate drops to a steady-state level (hopefully, quite low) for some period of time.
- As time passes, however, the failure rate rises again as hardware components suffer from the cumulative effects of dust, vibration, abuse, temperature extremes, and many other environmental maladies.

# Failure curve for hardware



# Failure curves for software



# Characteristics of Software

- Software is not susceptible to the environmental maladies that cause hardware to wear out.
- The failure rate curve for software should take the form of the “idealized curve” shown in Figure 1.2.
- Undiscovered defects will cause high failure rates early in the life of a program.
- These are corrected and the curve flattens as shown. The idealized curve is a gross oversimplification of actual failure models for software. However, the implication is clear—software doesn’t wear out. But it does *deteriorate!*

# Characteristics of Software

- This seeming contradiction can best be explained by considering the actual curve in Figure 1.2.
- During its life software will undergo change. As changes are made, it is likely that errors will be introduced, causing the failure rate curve to spike as shown in the “actual curve” (Figure 1.2).
- Before the curve can return to the original steady-state failure rate, another change is requested, causing the curve to spike again.
- Slowly, the minimum failure rate level begins to rise—the software is deteriorating due to change.

# Characteristics of Software

- Another aspect of wear illustrates the difference between hardware and software.
- When a hardware component wears out, it is replaced by a spare part.
- There are no software spare parts. Every software failure indicates an error in design or in the process through which design was translated into machine executable code.
- Therefore, the software maintenance tasks that accommodate requests for change involve considerably more complexity than hardware maintenance.

# Characteristics of Software

- ***Although the industry is moving toward component-based construction, most software continues to be custom-built.***
- As an engineering discipline evolves, a collection of standard design components is created.
- Standard screws and off-the-shelf integrated circuits are only two of thousands of standard components that are used by mechanical and electrical engineers as they design new systems.
- The reusable components have been created so that the engineer can concentrate on the truly innovative elements of a design, that is, the parts of the design that represent something new.
- In the hardware world, component reuse is a natural part of the engineering process.

# Characteristics of Software

- A software component should be designed and implemented so that it can be reused in many different programs. Modern reusable components encapsulate both data and the processing that is applied to the data, enabling the software engineer to create new applications from reusable parts.
- For example, today's interactive user interfaces are built with reusable components that enable the creation of graphics windows, pull-down menus, and a wide variety of interaction mechanisms.
- The data structures and processing detail required to build the interface are contained within a library of reusable components for interface construction.

# Software Application Domains

- system software
- application software
- engineering/scientific software
- embedded software
- product-line software
- WebApps (Web applications)
- AI software

- **system software**
- a collection of programs written to service other programs.
  - Some system software processes complex, but determinate, information structures.

    **e.g., compilers, editors, and file management utilities**
  - Other systems applications process largely indeterminate data.

    **e.g., operating system components, drivers, networking software, telecommunications processors**

- **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.
  - Application software is used to **control business functions in real time**
    - **e.g., point-of-sale transaction processing, real-time manufacturing process control.**

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

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

- **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**
- Address mass consumer markets
  - **e.g., word processing, spread sheets, computer graphics, multimedia, entertainment, database management, and personal and business financial applications.**

- **Web applications**—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.**
  - As Web 2.0 emerges, Web-Apps 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**.

- **Artificial intelligence software**—makes use of non - numerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis.
- Eg: Applications within this area include
  - robotics,
  - expert systems,
  - pattern recognition (image and voice),
  - artificial neural networks,
  - theorem proving,
  - and game playing.

# Legacy Software

## *Why must it change?*

- software must be adapted to meet the needs of new computing environments or technology.
- software must be enhanced to implement new business requirements.
- software must be extended to make it interoperable with other more modern systems or databases.
- software must be re-architected to make it viable within a network environment.

# Nature of WebApps - I

- **Network intensiveness.** A WebApp resides on a network and must *serve the needs* of a diverse community of clients.
- **Concurrency.** A large number of *users may access* the WebApp at one time.
- **Unpredictable load.** The number of users of the WebApp may *vary by orders of magnitude* from day to day.
- **Performance.** If a WebApp user must *wait too long* (for access, for server-side processing, for client-side formatting and display), he or she may decide to go elsewhere.
- **Availability.** Although expectation of 100 percent availability is unreasonable, users of popular WebApps often demand access on a *"24/7/365"* basis.

# Nature of WebApps - II

- **Data driven.** The primary function of many WebApps is *to use hypermedia* to present text, graphics, audio, and video content to the end-user.
- **Content sensitive.** The *quality and aesthetic* nature of *content* remains an important determinant of the *quality of a WebApp*.
- **Continuous evolution.** Unlike conventional application software that evolves over a series of planned, chronologically-spaced releases, Web applications evolve *continuously*.
- **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*.
- **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.
- **Aesthetics.** An undeniable part of the appeal of a WebApp is its *look and feel*.

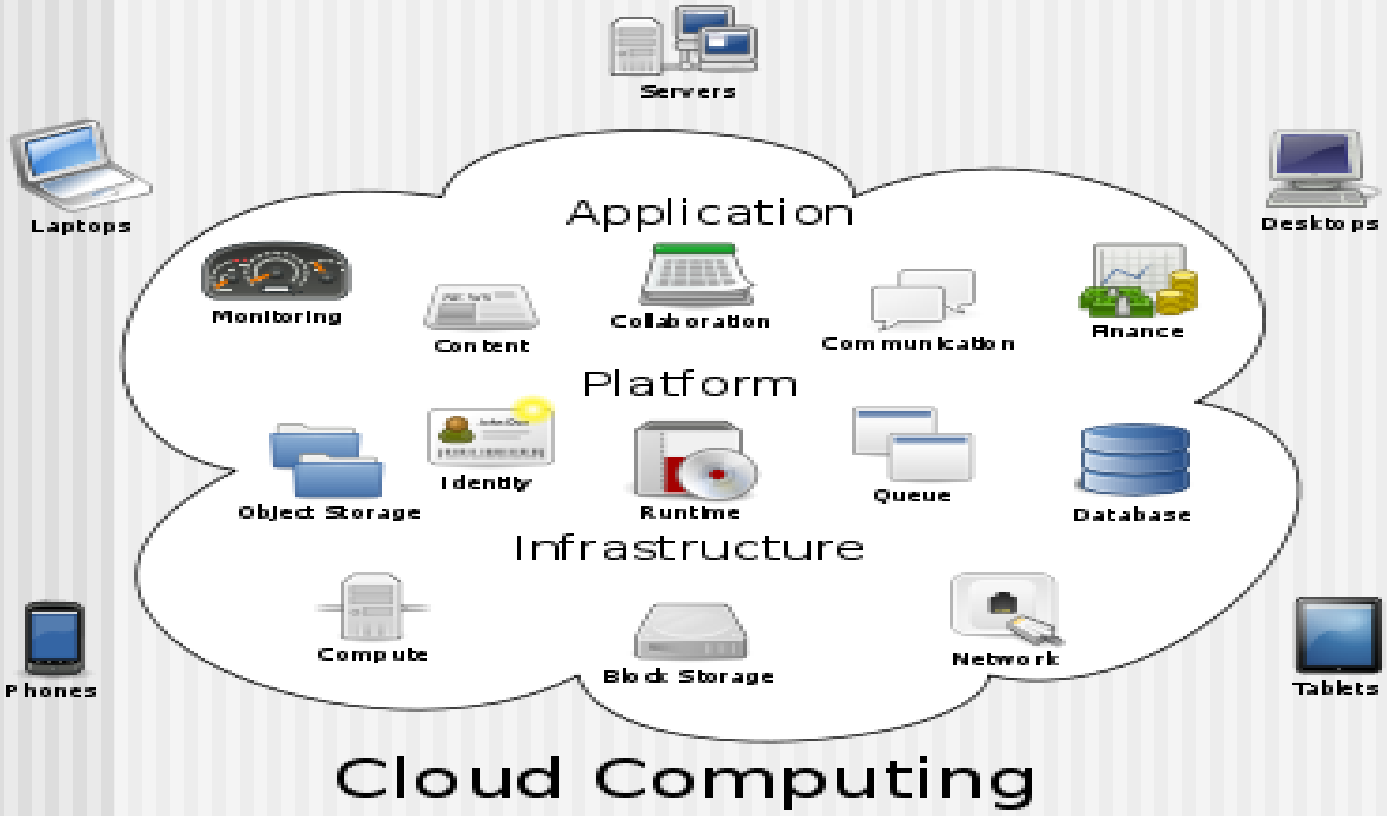
# The Changing Nature of Software
# 1. WebApps

- Modern WebApps are much more than hypertext files with a few pictures
- WebApps are augmented with tools like XML and Java to allow Web engineers including interactive computing capability
- WebApps may standalone capability to end users or may be integrated with corporate databases and business applications
- Semantic web technologies (Web 3.0) have evolved into sophisticated corporate and consumer applications that encompass semantic databases that require web linking, flexible data representation, and application programmer interfaces (API's) for access
- The aesthetic nature of the content remains an important determinant of the quality of a WebApp.

# 2. Mobile Apps

- Reside on mobile platforms such as cell phones or tablets
- Contain user interfaces that take both device characteristics and location attributes
- Often provide access to a combination of web-based resources and local device processing and storage capabilities
- Provide persistent storage capabilities within the platform
- A *mobile web application* allows a mobile device to access to web-based content using a browser designed to accommodate the strengths and weaknesses of the mobile platform
- A *mobile app* can gain direct access to the hardware found on the device to provide local processing and storage capabilities
- As time passes these differences will become blurred

# 3. Cloud Computing



# 3. Cloud Computing

- *Cloud computing* provides distributed data storage and processing resources to networked computing devices
- Computing resources reside outside the cloud and have access to a variety of resources inside the cloud
- Cloud computing requires developing an architecture containing both frontend and backend services
- Frontend services include the client devices and application software to allow access
- Backend services include servers, data storage, and server-resident applications
- Cloud architectures can be segmented to restrict access to private data

# 4. Product Line Software

- *Product line software* is a set of software-intensive systems that share a common set of features and satisfy the needs of a particular market
- These software products are developed using the same application and data architectures using a common core of reusable software components
- A software product line shares a set of assets that include *requirements, architecture, design patterns, reusable components, test cases,* and other work products
- A software product line allow in the development of many products that are engineered by capitalizing on the commonality among all products with in the product lin

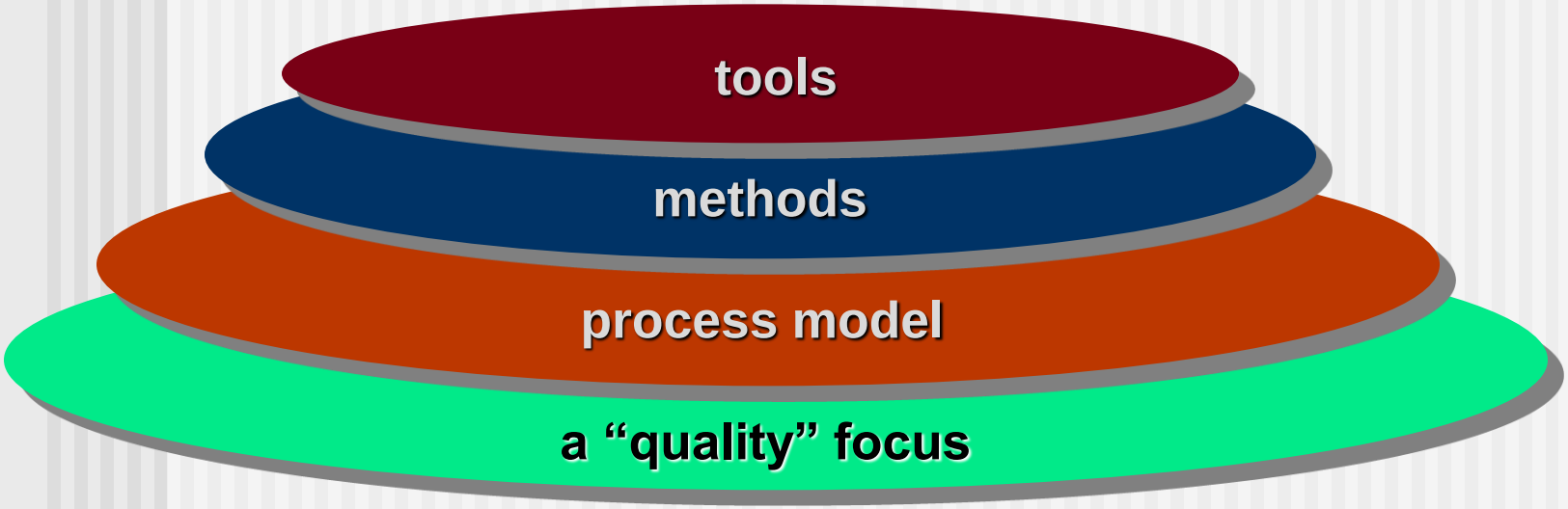
# Software Engineering

- Some realities:
  - *a concerted effort should be made to understand the problem before a software solution is developed*
  - *design becomes a pivotal activity*
  - *software should exhibit high quality*
  - *software should be maintainable*
- The seminal definition:
  - *[Software engineering is] the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.*

# Software Engineering

- The IEEE definition:
  - *Software Engineering: (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1).*

# A Layered Technology




***Software Engineering***

# A Layered Technology

- Software engineering is a layered technology.
- Referring to Figure (previous slide), software engineering must rest on an organizational commitment to *quality*.
- Total quality management, Six Sigma, and similar philosophies foster a continuous process improvement culture, leads to the development of increasingly more effective approaches to software engineering.
- The bedrock that supports software engineering is a quality focus.

# A Layered Technology

- The foundation for software engineering is the *process* layer.
- The software engineering **process** is the glue that holds the **technology layers** together and enables **timely development** of computer software.
- **Process defines a framework** that must be established for **effective delivery** of software engineering technology.
- The **software process** forms the basis for management control of software projects and establishes the context in which

  **technical methods are applied,**

  **work products (models, documents, data, reports,**

  **forms, etc.) are produced, milestones are established, quality is ensured, change is properly managed**

# A Layered Technology

- Software engineering ***methods*** provide the technical how-to's for building software.
- Methods encompass a broad array of tasks that include

  **communication,**

  **requirements analysis,**

  **design modelling,**

  **program construction,**

  **testing, and support**.
- Software engineering methods rely on a set of basic principles that govern each area of the technology and include modelling activities and other descriptive techniques.

# A Layered Technology

- Software engineering ***tools*** provide automated or semi-automated support for the process and the methods.
- When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called *computer-aided software engineering*, is established.

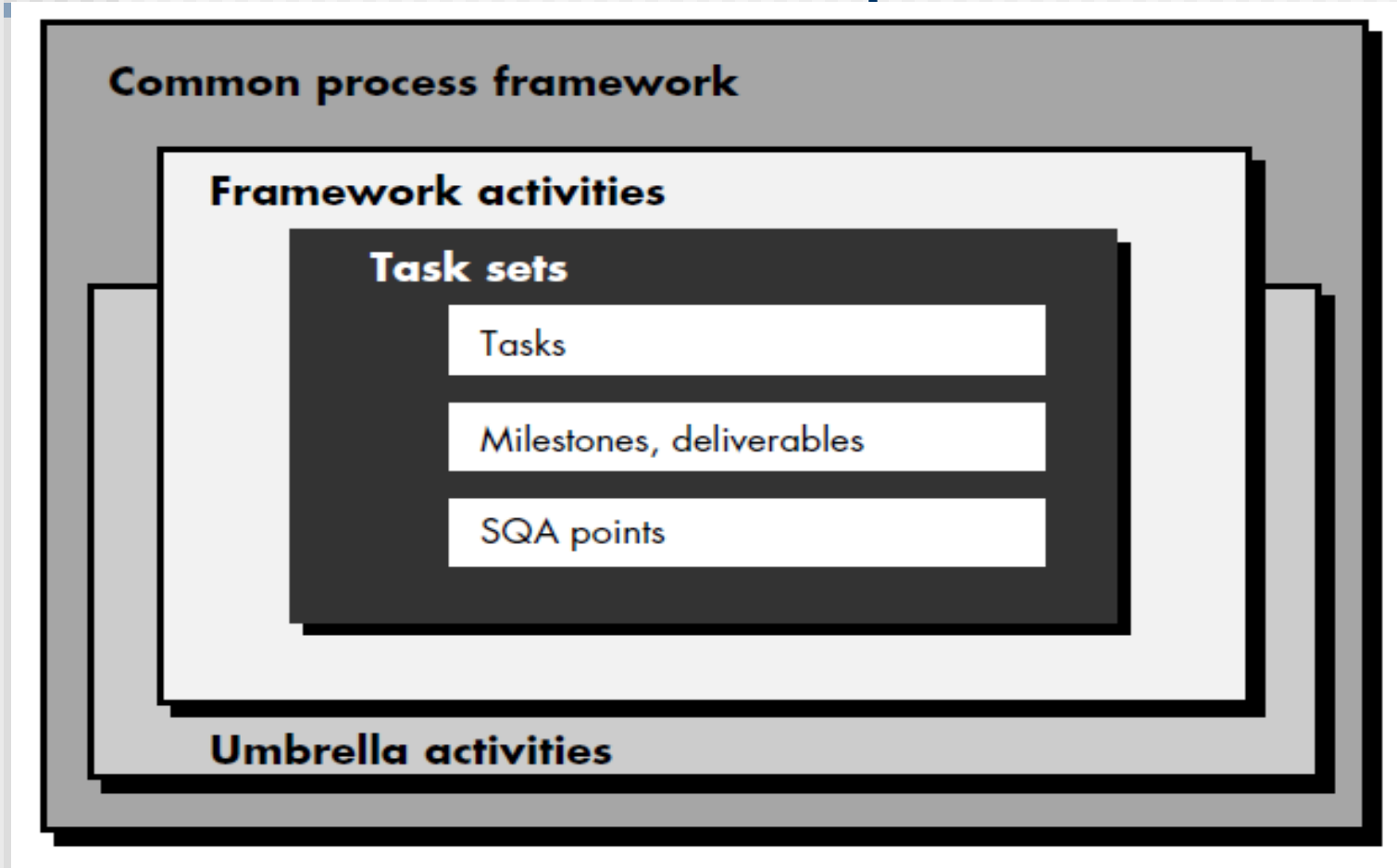
# Generic process Framework Activities

- Communication
- Planning
- Modeling
  - Analysis of requirements
  - Design
- Construction
  - Code generation
  - Testing
- Deployment

# Umbrella Activities

- Software project management
- Formal technical reviews
- Software quality assurance
- Software configuration management
- Work product preparation and production
- Reusability management
- Measurement
- Risk management

# The software process



# A Process Framework

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* and is applied regardless of the application domain, size of the project, complexity of the effort, or degree of rigor with which software engineering is to be applied.
**e.g., communication with stakeholders**

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

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

# A Process Framework

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

The ***intent*** is always to deliver software in a timely manner and with sufficient quality to satisfy those who have sponsored its creation and those who will use it.

- **Communication.** Before any technical work can commence, customer communicate and collaborate with other stakeholders.

The intent is to *understand stakeholders' objectives* for the project and to *gather requirements* that help define software features and functions.

# A Process Framework

- **Planning.** 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**.

- **Modelling.** It creates a “sketch” of the thing—what it will look like architecturally, how the constituent parts fit together, and many other characteristics.

If required, refine the sketch into *greater and greater detail* in an effort to better understand the problem and how 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.

# A Process Framework

- **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.

That is, **communication, planning, modelling, construction,** and **deployment** are applied repeatedly through a number of project iterations.

# A Process Framework

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

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 ensure 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.

# A Process Framework

**Measurement**—defines and collects **process, project, and product measures** that assist the team in delivering software that meets stakeholders' needs; can be used in conjunction with all other framework and umbrella activities.

**Software configuration management**—manages the effects of change throughout the software process.

**Reusability management**—defines criteria for work product reuse (including software components) and establishes mechanisms to achieve reusable components.

**Work product preparation and production**—encompasses the activities required to create work products such as models, documents, logs, forms, and lists.

# Process Adaptation

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

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

Among the differences are

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

# Software Engineering practice

- The generic software process model composed of a set of activities that establish a framework for software engineering practice.
- Generic framework activities—**communication, planning, modelling, construction,** and **deployment**—and umbrella activities establish a skeleton architecture for software engineering work.
- But how does the practice of software engineering fit in?
- the generic concepts and principles that apply to framework activities.

# The Essence of Practice

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

# Understand the Problem

- *Who has a stake in the solution to the problem?* That is, **who are the stakeholders?**
- *What are the unknowns?* **What data, functions, and features are required to** properly **solve** the problem?
- *Can the problem be compartmentalized?* Is it possible to **represent smaller problems** that may be easier to understand?

# Plan the Solution

- *Have you seen similar problems before?* Are there patterns that are recognizable in a potential solution? **Is there existing software that implements the data, functions, and features that are required?**
- *Has a similar problem been solved?* If so, **are elements of the solution reusable?**
- *Can subproblems be defined?* If so, **are solutions readily apparent for the subproblems?**
- *Can you represent a solution in a manner that leads to effective implementation?* **Can a design model be created?**

# Carry Out the Plan

- *Does the solution conform to the plan?* Is source code **traceable to the design model**?

- *Is each component part of the solution provably correct?* Has the **design and code been reviewed,** or better, have **correctness proofs been applied to algorithm?**

# Examine the Result

- *Is it possible to test each component part of the solution?* Has a reasonable **testing strategy been implemented?**
- *Does the solution produce results that conform to the data, functions, and features that are required?* Has the **software** been **validated** against all stakeholder **requirements**?

# Hooker's General Principles

- 1: *The Reason It All Exists*
- 2: *KISS (Keep It Simple, Stupid!)*
- 3: *Maintain the Vision*
- 4: *What You Produce, Others Will Consume*
- 5: *Be Open to the Future*
- 6: *Plan Ahead for Reuse*
- 7*: Think!*

# Hooker's General Principles

- **1: *The Reason It All Exists***
- A software system exists for one reason: ***to provide value to its users*.**
- Ask questions such as: "Does this add real value to the system?" If the answer is "no," don't do it. All other principles support this one.
- All decisions should be made with this in mind.
  - **Before specifying a system requirement,**
  - **before noting a piece of system functionality,**
  - **before determining the hardware platforms or development processes,**

# Hooker's General Principles

- **2: *KISS (Keep It Simple, Stupid!)***
- Software design is not a haphazard process. There are many factors to consider in any design effort. ***All design should be as simple as possible, but no simpler*.**
- This facilitates having a more easily understood and easily maintained system. **Features, or internal features, should not be discarded in the name of simplicity.**
- Indeed, the more elegant designs are usually the more simple ones. Simple also **does not mean "quick and dirty."**
- It takes a lot of thought and work over multiple iterations to simplify**. The payoff is software that is more maintainable and less error-prone**.

# Hooker's General Principles

- **3: *Maintain the Vision***
- *A clear vision is essential to the success of a software project*. Without one, a project almost unfailingly ends up being "of two [or more] minds" about itself.
- Without conceptual integrity, a system threatens to become a patchwork of incompatible designs.
- **Compromising the architectural vision of a software system weakens and will eventually break even the well-designed systems.**
- **Having an empowered architect who can hold the vision and enforce compliance helps ensure a very successful software project.**

# Hooker's General Principles

- **4: *What You Produce, Others Will Consume***
- *Always specify, design, and implement knowing someone else will have to understand what you are doing*. The audience for any product of software development is potentially large.
- Specify with an eye to the users.
- Design, keeping the implementers in mind.
- Code with concern for those that must maintain and extend the system.
- Someone may have to debug the code you write, and that makes them a user of your code. Making their job easier adds value to the system.

# Hooker's General Principles

- **5: *Be Open to the Future***
- A system with a long lifetime has more value.
- In today's computing environments, where specifications change on a moment's notice and hardware platforms are obsolete just a few months old, **software lifetimes are typically measured in months instead of years**. However, true "industrial-strength" software systems must endure far longer.
- To do this successfully, these systems must be ready to adapt to these and other changes. Systems that do this successfully are those that have been designed this way from the start.
- ***Never design yourself into a corner*. Always ask "what if," and prepare for all possible answers by creating systems that solve the general problem, not just the specific one.**.

# Hooker's General Principles

- **6: *Plan Ahead for Reuse***
- **Reuse saves time and effort.**
- Achieving a high level of reuse is arguably the hardest goal to accomplish in developing a software system.
- The reuse of code and designs has been proclaimed as a major benefit of using **object-oriented technologies.**
- The return on this investment is not automatic. To leverage the reuse possibilities that object-oriented [or conventional] programming provides **requires forethought and planning.**
- There are many techniques to realize reuse at every level of the system development process. . . . ***Planning ahead for reuse reduces the cost and increases the value of both the reusable components and the systems into which they are incorporated*.**

# Hooker's General Principles

- ***7: Think!***
- *Placing clear, complete thought before action almost always produces better results*.
- When you think about something, you are more likely to do it right.
- **You also gain knowledge about how to do it right again. If you do think about something and still do it wrong, it becomes a valuable experience**.
- A side effect of thinking is **learning to recognize** when you don't know something, at which point you can research the answer. **When clear thought has gone into a system, value comes out.**
  - **If every software engineer and every software team simply followed Hooker's seven principles, many of the difficulties we experience in building complex computer based systems would be eliminated.**

# Software Myths

- Affect managers, customers (and other non-technical stakeholders) and practitioners
- Are believable because they often have elements of truth,

*but …*

- Invariably lead to bad decisions,

*therefore …*

- Insist on reality as you navigate your way through software engineering

# Software Myths

**Definition**: Erroneous beliefs about software and the process used to build it. Myths have number of attributes that have made them insidious (i.e. dangerous).

- Misleading Attitudes - caused serious problem for managers and technical people.

## Management myths

Managers in most disciplines, are often under pressure to maintain budgets, keep schedules on time, and improve quality.

**Myth1**: We already have a book that's full of standards and procedures for building software, won't that provide my people with everything they need to know?

**Reality** :

- Are software practitioners aware of existence standards?
- Does it reflect modern software engineering practice?
- Is it complete? Is it streamlined to improve time to delivery while still maintaining a focus on quality?

**Myth2*:*** If we get behind schedule, we can add more programmers and catch up

**Reality*:*** Software development is not a mechanistic process like manufacturing. Adding people to a late software project makes it later.

- People can be added but only in a planned and well-coordinated manner

**Myth3*:*** If I decide to outsource the software project to a third party, I can just relax and let that firm build it.

**Reality***:* If an organization does not understand how to manage and control software projects internally, it will invariably struggle when it outsource software projects

# Customer Myths

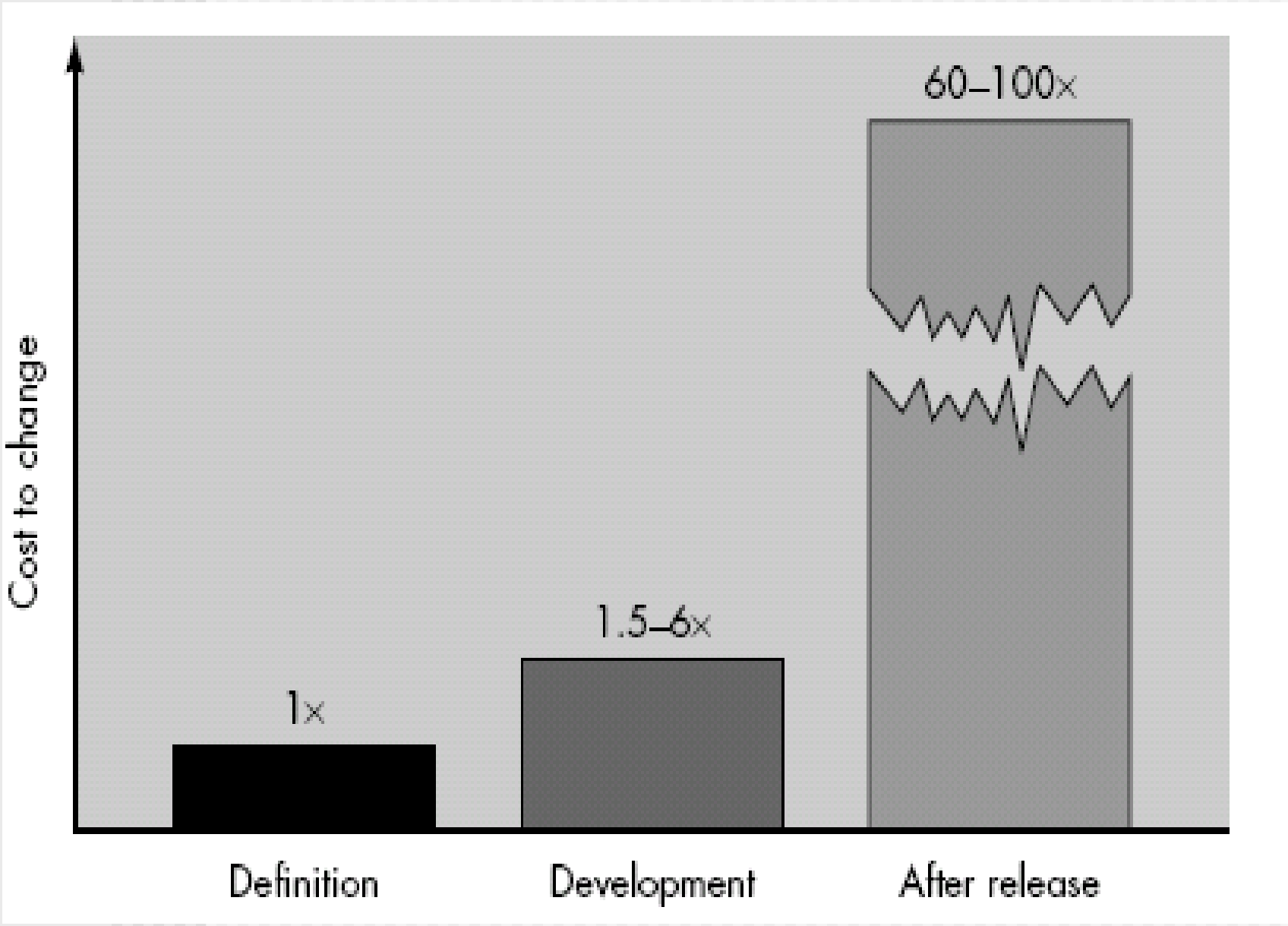
Customer may be a person from inside or outside the company that has requested software under contract.

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

**Reality*:*** A poor up-front definition is the major cause of failed software efforts. A formal and detailed description of the information domain, function, behavior, performance, interfaces, design constraints, and validation criteria is essential. These characteristics can be determined only after thorough communication between customer and developer.

**Myth*:*** Project requirements continually change, but change can be easily accommodated because software is flexible.

**Reality:** Customer can review requirements and recommend modifications with relatively little impact on cost. When changes are requested during software design, the cost impact grows rapidly. Below mentioned *figure* for reference.


Cost to change
60–100×
1.5–6×
1×
Definition
Development
After release

# Practitioner's myths

**Myth1*:*** Once we write the program and get it to work, our job is done.

**Reality*:*** Someone once said that "the sooner you begin 'writing code', the longer it'll take you to get done." Industry data indicate that between 60 and 80 percent of all effort expended on software will be expended after it is delivered to the customer for the first time.

**Myth2*:*** Until I get the program "running" I have no way of assessing its quality.

**Reality:** One of the most effective software quality assurance mechanisms can be applied from the inception of a project—the *formal technical review.*

*Software reviews* are a "**quality filter**" that have been found to be more effective than testing for finding certain classes of software defects.

**Myth3:** The only deliverable work product for a successful project is the working program.

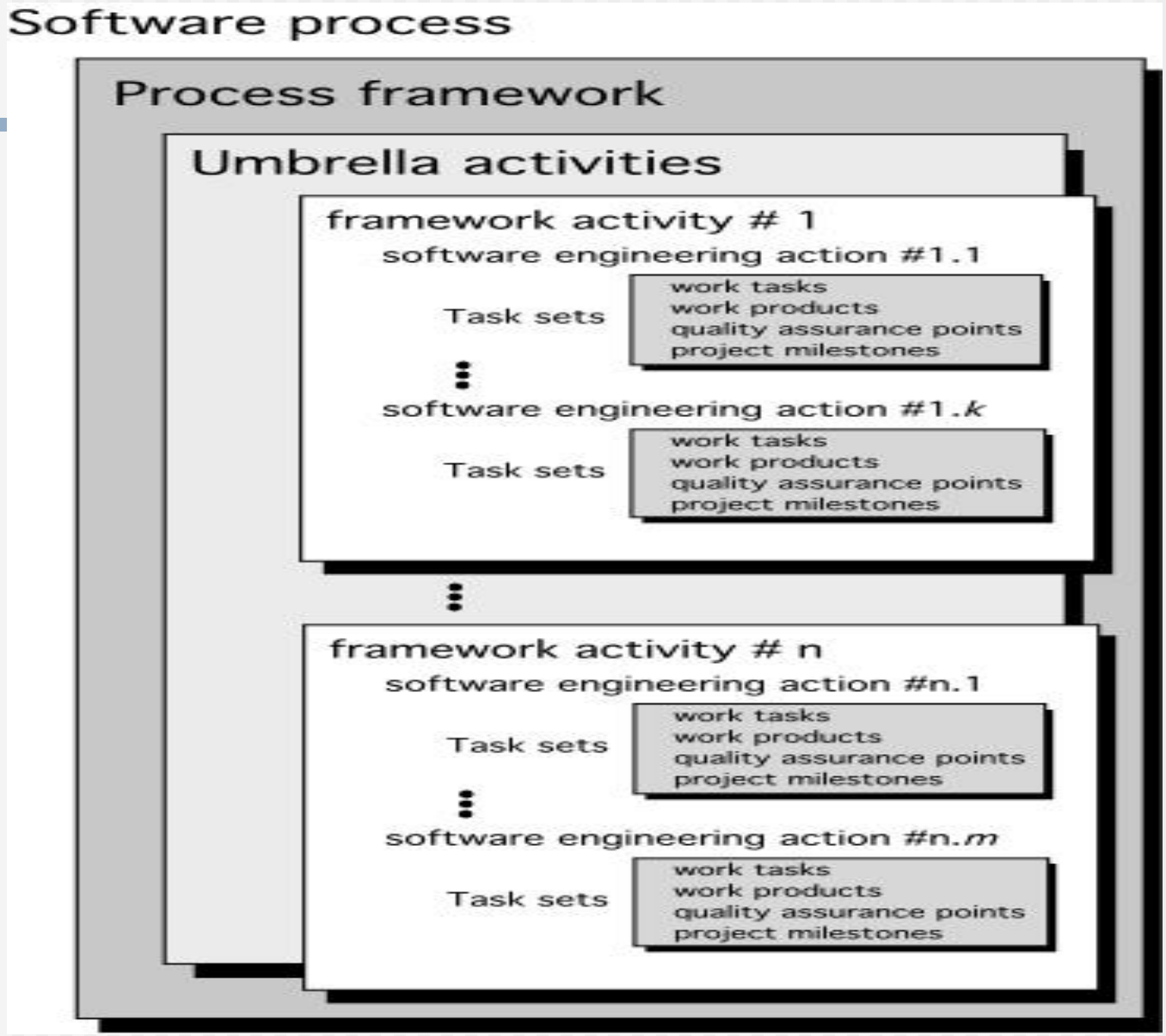
**Reality*:*** A working program is only one part of a *software configuration* that includes many elements.

Models, documentation, plans provides a foundation for successful engineering and, guidance for software support.

**Myth4 *:*** Software engineering will make us create voluminous and unnecessary documentation and will invariably slow us down.

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

# A Generic Process Model



# A Generic Process Model

- The software process is represented schematically in Figure 2.1.
- In each framework activity is populated by a set of software engineering actions.
- Each software engineering action is defined by a ***task set*** that identifies
  - the **work tasks** that are **to be completed**,
  - the **work products** that will **be produced**,
  - the **quality assurance points** that will be **required**,
  - and the **milestones** that will be used to **indicate progress**.

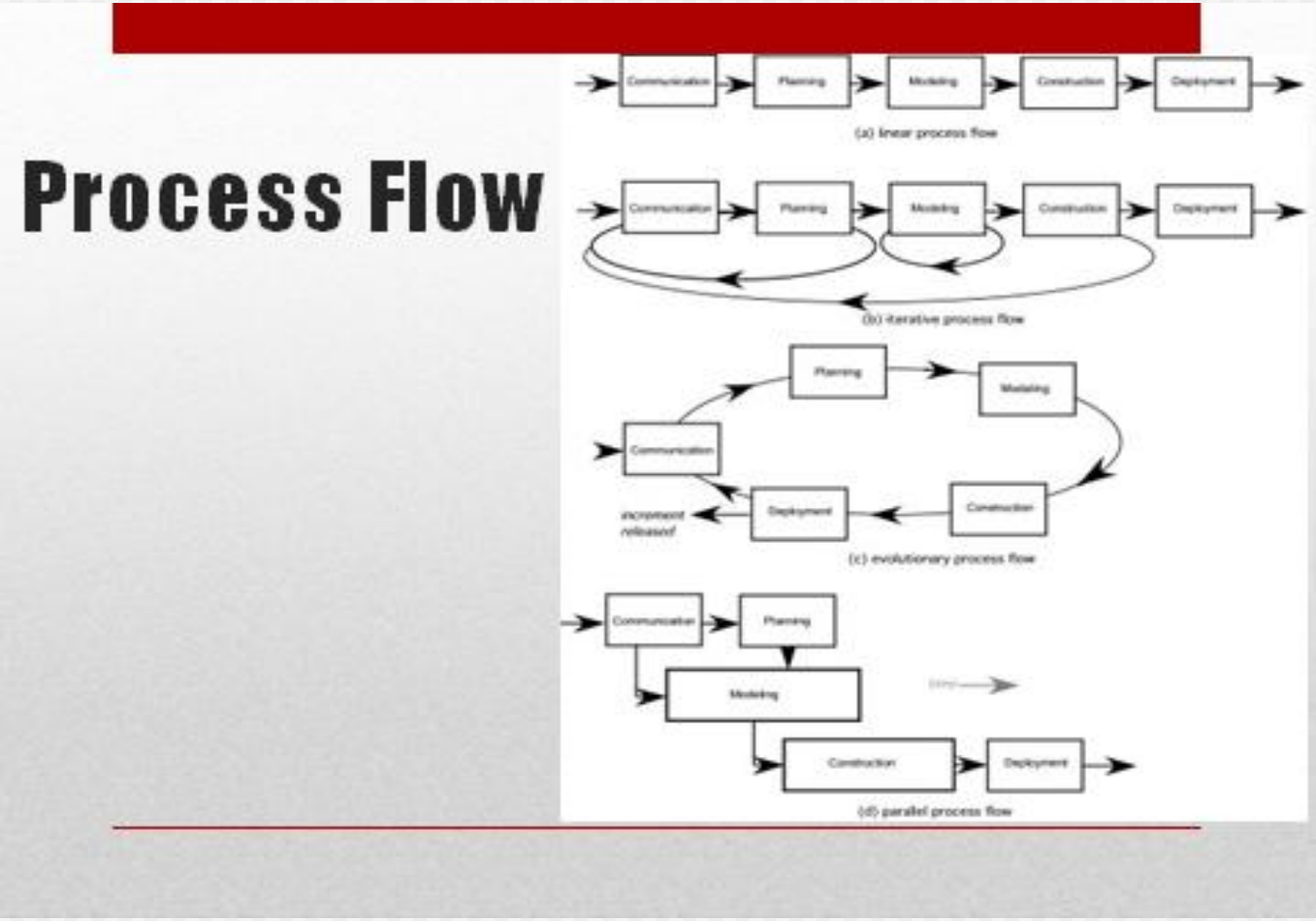
# A Generic Process Model

A **generic process framework for software engineering defines** five framework activities**—communication, planning, modeling, construction,** and **deployment.**

A set of **umbrella activities—project tracking and control, risk management, quality assurance, configuration management, technical reviews**, and others—are applied throughout the process.

***process flow*—describes how the framework activities and the actions and tasks that occur within each framework activity are organized with respect to sequence and time**

# Process Flow



# A Generic Process Model

- A ***linear process flow*** executes each of the five framework activities in sequence, beginning with communication and culminating with deployment (Figure 2.2a).
- An ***iterative process flow*** repeats one or more of the activities before proceeding to the next (Figure 2.2b).
- An ***evolutionary process flow*** executes the activities in a "circular" manner. Each circuit through the five activities leads to a more complete version of the software (Figure 2.2c).
- A ***parallel process flow*** (Figure 2.2d) executes one or more activities in parallel with other activities (e.g., modelling for one aspect of the software might be executed in parallel with construction of another aspect of the software).

# A Generic Process Model

A software team would need significantly more information before it could properly execute any one of these activities as part of the software process.
A key question:

***What actions are appropriate for a framework activity, given the nature of the problem to be solved, the characteristics of the people doing the work, and the stakeholders who are sponsoring the project?***

# Identifying a Task Set

- For a small software project requested by one person (at a remote location) with simple, straightforward requirements, the communication activity might encompass little more than a phone call with the appropriate stakeholder. Therefore, the only necessary action is *phone conversation,* and the work tasks (the *task set*) that this action encompasses are:
- **1. Make contact with stakeholder via telephone.**
- **2. Discuss requirements and take notes.**
- **3. Organize notes into a brief written statement of requirements.**
- **4. E-mail to stakeholder for review and approval.**

# Identifying a Task Set

- A task set defines the actual work to be done to accomplish the objectives of a software engineering action.
  - **A list of the task to be accomplished**
  - **A list of the work products to be produced**
  - **A list of the quality assurance filters to be applied**

- Referring again to Figure 2.1, each software engineering action (e.g., *elicitation,* an action associated with the communication activity) can be represented by a number of different *task sets*—each a collection of software engineering work tasks, related work products, quality assurance points, and project milestones.

# Process Patterns

- A *process pattern*
  - describes a process-related problem that is encountered during software engineering work,
  - identifies the environment in which the problem has been encountered, and
  - suggests one or more proven solutions to the problem.
- Stated in more general terms, a process pattern provides you with a *template*—a consistent method for describing problem solutions within the context of the software process.

# Process Patterns

- **Pattern Name.** The pattern is given a **meaningful name describing it within the context of the software process**
  - (e.g., **TechnicalReviews**).

- **Forces.** The **environment** in which the pattern is **encountered** and the issues that make the problem visible and may affect its solution.

# Process Pattern Types

- **Type.** The pattern type is specified. Ambler suggests three types:
- ***Stage patterns*—defines a problem** associated with a **framework activity** for the process.
- ***Task patterns*—defines a problem** associated with **a software engineering action or work task** and relevant to successful software engineering practice
- ***Phase patterns*—define the sequence of framework activities** that occur with the process, even when the overall flow of activities is iterative in nature.

# Process Pattern Types

- **Initial context.** Describes the conditions under which the pattern applies. Prior to the initiation of the pattern:
  - **(1) What organizational or team-related activities have already occurred?**
  - **(2) What is the entry state for the process?**
  - **(3) What software engineering information or project information already exists?**
  - For example, the **Planning** pattern (a stage pattern) requires that
    - ✓ (1) Customers and software engineers have **established a collaborative communication;**
    - ✓ (2) **Successful completion of a number of task patterns** [specified] for the **Communication** pattern has occurred; and
    - ✓ (3) The **project scope, basic business requirements, and project constraints are kno**wn.

# Process Pattern Types

- **Problem.** The **specific problem to be solved** by the pattern.

- **Solution.** Describes **how to implement the pattern successfully.**
  - This **describes how the initial state of the process** (that exists before the pattern is implemented) **is modified** as a consequence of **the initiation of the pattern**.
  - It also describes **how software engineering information or project information that is available before the initiation of the pattern is transformed** as a consequence of the **successful execution of the pattern.**

# Process Pattern Types

- **Resulting Context. Describes the conditions that will result once the pattern has been successfully implemented.** Upon completion of the pattern:
  - (1) What **organizational or team-related activities** must have **occurred**?
  - (2) What is the **exit state** for the process?
  - (3) What **software engineering information or project information** has been developed?
- **Related Patterns.** Provide a list of all process **patterns that are directly related** to this one. This may be **represented as a hierarchy or in some other diagrammatic form**.
  - For example, the stage pattern **Communication** encompasses the task patterns: **Project Team, Collaborative Guidelines, Scope Isolation, Requirements Gathering, Constraint Description,** and **Scenario Creation.**

# Process Pattern Types

- **Known Uses and Examples.** Indicate the specific instances in which the pattern is applicable. For example, **Communication** is mandatory at the beginning of every software project, is recommended throughout the software project, and is mandatory once the deployment activity is under way.

# Process Assessment and Improvement

- **Standard CMMI Assessment Method for Process Improvement (SCAMPI)** — provides a **five step process assessment model that incorporates five phases: initiating, diagnosing, establishing, acting and learning.**
- **CMM-Based Appraisal for Internal Process Improvement (CBA IPI)**—provides a **diagnostic technique for assessing the relative maturity of a software organization**; uses the SEI CMM as the basis for the assessment [Dun01]
- **SPICE—The SPICE (ISO/IEC15504)** standard **defines a set of requirements for software process assessment**. The intent of the standard is to assist organizations in **developing an objective evaluation of the efficacy** of any defined software process. [ISO08]
- **ISO 9001:2000 for Software—**a generic standard that applies to any organization that wants to improve **the overall quality of the products, systems, or services that it provides.** Therefore, the standard is directly applicable to software organizations and companies. [Ant06]

# Prescriptive Models

- Prescriptive process models advocate an orderly approach to software engineering

*That leads to a few questions …*

- If prescriptive process models strive for structure and order, are they inappropriate for a software world that thrives on change?
- Yet, if we reject traditional process models (and the order they imply) and replace them with something less structured, do we make it impossible to achieve coordination and coherence in software work?

# Prescriptive Models

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

# Prescriptive Models

- The *waterfall model,* sometimes called the ***classic life cycle***, suggests a **systematic, sequential approach** to software development that begins with customer specification of requirements and progresses through planning, modelling, construction, and deployment, culminating in on going support of the completed software (Figure A).
- A variation in the representation of the waterfall model is called the *V-model*. Represented in Figure B, the V-model depicts the **relationship of quality assurance actions** to the **actions associated with communication, modelling, and early construction activities.**
- As a software **team moves down the left side** of the V, basic **problem requirements are refined** into progressively more detailed and technical representations of the problem and its solution.

# The Waterfall Model

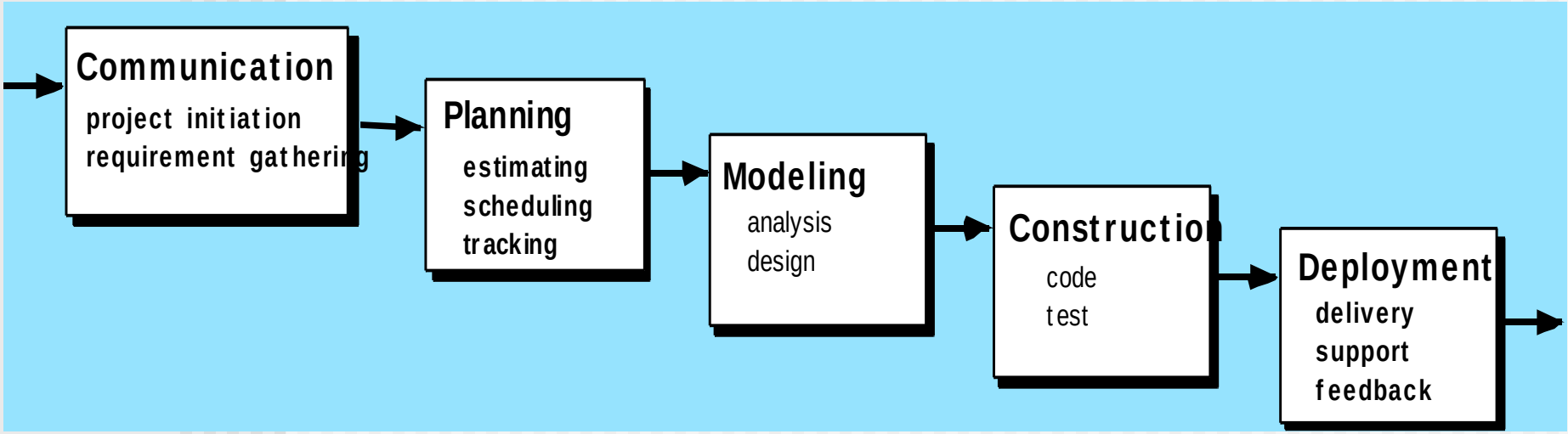



Fig A

# The V-Model

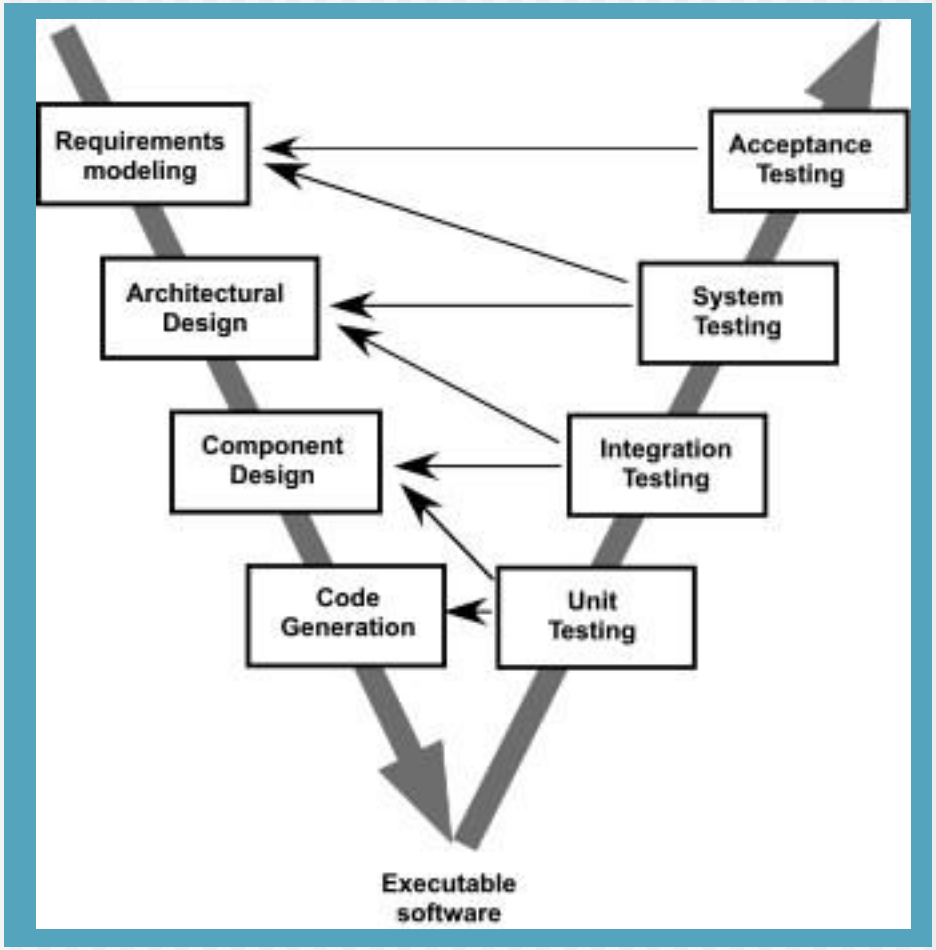



Fig B

## Prescriptive Models The Waterfall Model

- **Once code has been generated, the team moves up the right side of the V, essentially performing a series of tests (quality assurance actions) that validate each of the models created as the team moved down the left side.**
- There is no fundamental difference between the classic life cycle and the V-model.
- The **V-model provides a way of visualizing how verification and validation actions** are applied to earlier engineering work.
- The waterfall model is the oldest paradigm for software engineering.
- Over the past three decades, criticism of this process model has caused even ardent supporters to question its efficacy.

# Prescriptive Models The Waterfall Model

- ***Disadvantages of Waterfall Model***
- **1.** Real projects rarely follow the sequential flow that the model proposes. Although **the linear model can accommodate iteration, *it does so indirectly*. As a result, changes can cause confusion as the project team proceeds.**
- **2.** It is often **difficult for the customer to state all requirements explicitly**. The waterfall model requires this and has difficulty accommodating the natural uncertainty that exists at the beginning of many projects.
- **3.** The **customer must have patience**. A **working version of the program(s) will not be available until late in the project time span.** A major blunder, if undetected until the working program is reviewed, can be disastrous.

## Prescriptive Models The Waterfall Model

- **Bradac** found that the linear nature of the classic life cycle leads to **"blocking states"** in which some project team members must wait for other members of the team to complete dependent tasks.
- The time spent waiting can exceed the time spent on productive work!
- The blocking states tend to be more prevalent at the **beginning and end of a linear sequential process.**
- **Today, software work is fast-paced and subject to a never-ending stream of changes (to features, functions, and information content).**
- **The waterfall model is often inappropriate for such work**.
- It can serve as a useful process model in situations where requirements are fixed and work is to proceed to completion in a linear manner.

# The Incremental Model

- There are many situations in which initial software requirements are reasonably well defined, but the overall scope of the development effort precludes a purely linear process.
- **There may be a compelling need to provide a limited set of software functionality to users quickly and then refine and expand on that functionality in later software releases.**
- In such cases, you can choose a process model that is designed to produce the software in increments.
- The *incremental* model **combines** elements of **linear and parallel process flows.**
- Referring to Figure 2.5, the incremental model applies linear sequences in a staggered fashion as calendar time progresses.
- Each **linear sequence produces deliverable "increments"** of the software in a manner that is similar to the increments produced by an evolutionary process flow

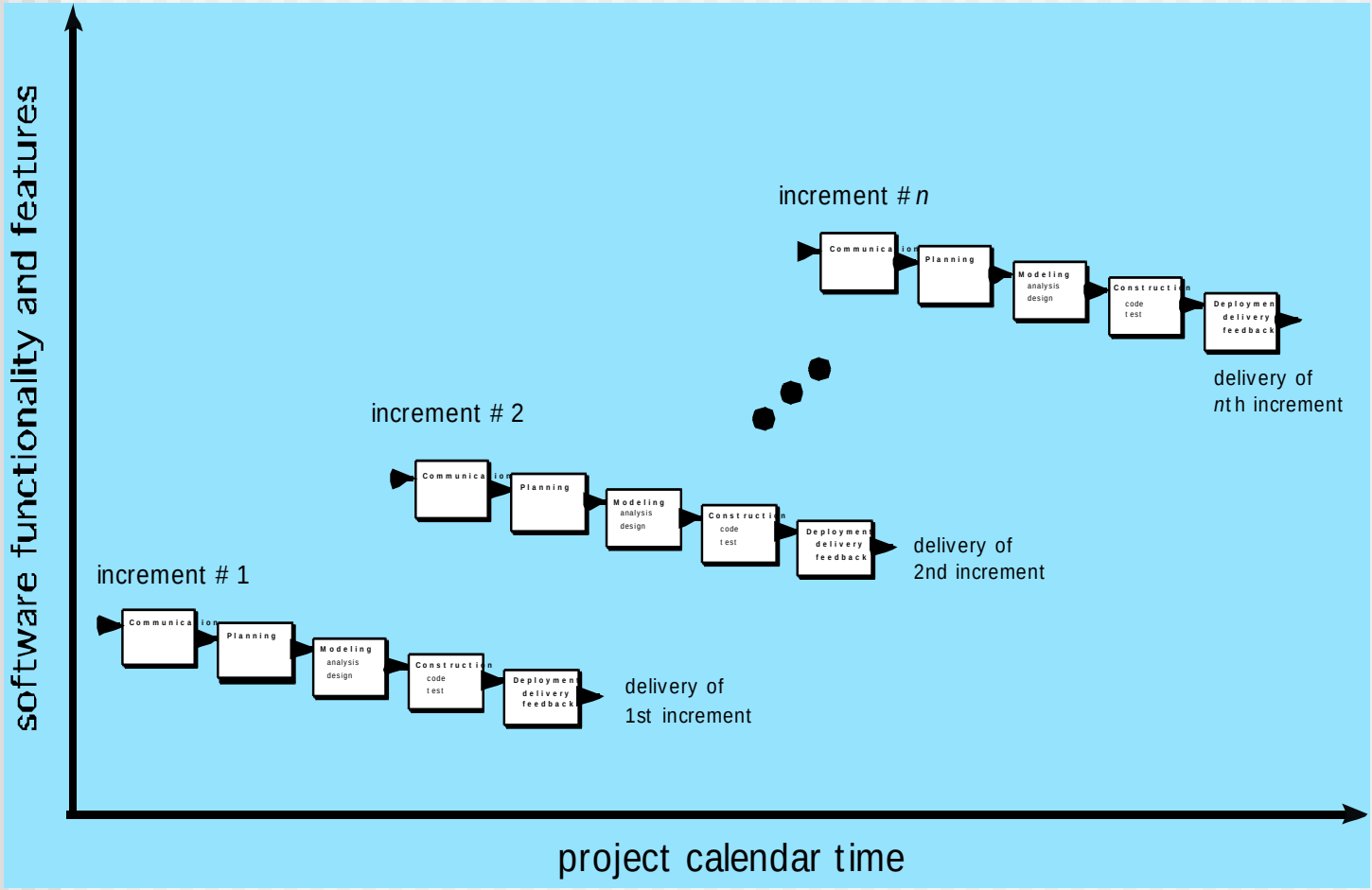
# The Incremental Model

- For example, word-processing software developed using the incremental paradigm might deliver
  - **basic file management, editing, and document production functions in the first increment;**
  - **more sophisticated editing and document production capabilities in the second increment;**
  - **spelling and grammar checking in the third increment;**
  - **and advanced page layout capability in the fourth increment. It should be noted that the process flow for any increment can incorporate the prototyping paradigm.**
- When an incremental model is used, the **first increment** is often a ***core product***.
- That is, basic requirements are addressed but many supplementary features (some known, others unknown) remain undelivered.

# The Incremental Model

- The **core product is used by the customer** (or undergoes detailed evaluation).
- **As a result of use and/or evaluation, a plan is developed for the next increment.**
- The **plan addresses the modification** of the core product to **better meet the needs** of the customer and the delivery of **additional features and functionality**.
- This process is repeated following the delivery of each increment, until the complete product is produced.

# The Incremental Model



# The Incremental Model

- The incremental process model **focuses on the delivery of an operational product with each increment**.
- Early increments are ***stripped-down versions*** of the final product, but they do **provide capability that serves the user and also provide a platform for evaluation by the user.**
- **Incremental development is particularly useful when *staffing is unavailable* for a complete implementation *by the business deadline* that has been established for the project**.
- Early increments can be implemented with fewer people. If the core product is well received, then additional staff (if required) can be added to implement the next increment.
- Increments can be planned to manage technical risks.
  - **For example, a major system might require the availability of new hardware that is under development and whose delivery date is uncertain.**
  - **It might be possible to plan early increments in a way that avoids the use of this hardware, thereby enabling partial functionality to be delivered to end users without inordinate delay.**

# Evolutionary Models: Prototyping

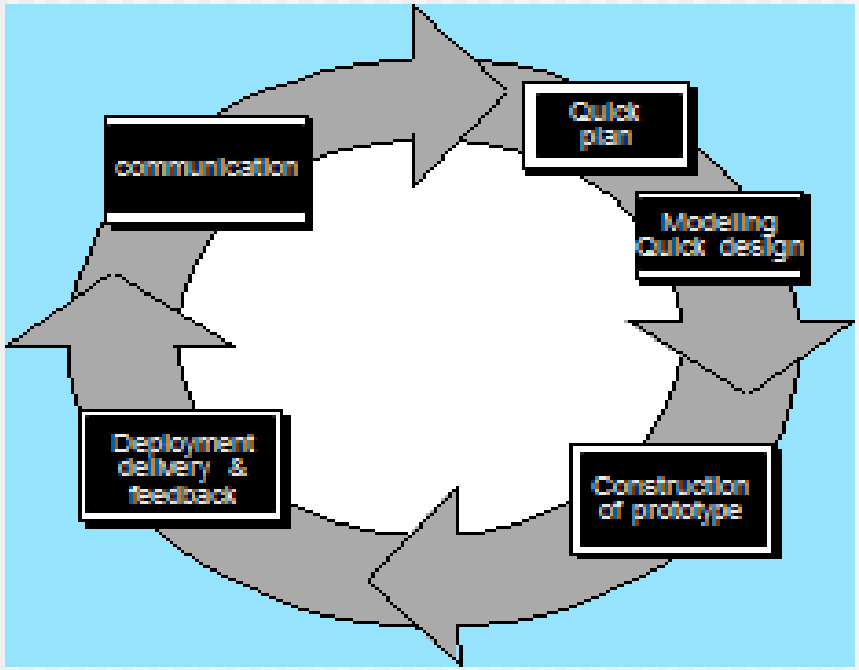
- Evolutionary models are iterative. They are characterized in a manner that enables you to develop increasingly more complete versions of the software.

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

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

# Evolutionary Models: Prototyping

## Evolutionary Models: Prototyping



# Evolutionary Models: Prototyping

- The prototyping paradigm (Figure above slide) begins with communication. You meet with other stakeholders to define the overall objectives for the software, **identify whatever requirements are known, and outline areas where further definition is mandatory.**
- A prototyping iteration is planned quickly, and modelling (in the form of a “quick design”) occurs.
- A quick design focuses on a representation of those aspects of the software that will be visible to end users (e.g., human interface layout or output display formats).
- The quick design leads to the construction of a prototype.
- The **prototype is deployed and evaluated by stakeholders**, who **provide feedback** that is used to further refine requirements.
- **Iteration occurs as the prototype is tuned to satisfy the needs of various stakeholders**, while at the same time enabling you to better understand what needs to be done.

# Evolutionary Models: Prototyping

- The prototype serves as a **mechanism for identifying software requirements.**
- If a working prototype is to be built, existing program fragments or apply tools (e.g., report generators and window managers) **can be used that enable working programs to be generated quickly.**
- But what do you do with the prototype when it has served the purpose described earlier? Brooks provides one answer:
- In most projects, the **first system built is barely usable. It may be too slow, too big, awkward in use or all three.** There is no alternative but to start again, smarting but smarter, and ***build a redesigned version in which these problems are solved.***
- The prototype can serve as **"the first system."** The one that Brooks recommends you throw away. But this may be an idealized view. Although **some** prototypes are built as "***throwaways*****,**" **others are evolutionary in the sense that the prototype slowly evolves into the actual system.**

# Evolutionary Models: Prototyping

- Both stakeholders and software engineers like the prototyping paradigm. Users get a feel for the actual system, and developers get to build something immediately. Yet, prototyping can be **problematic for the following reasons**:

- **1.** Stakeholders see what appears to be a working version of the software,
  - **unaware that the prototype is held together haphazardly**,
  - **unaware that in the rush to get it working you haven't considered overall software quality or long-term maintainability.**
- When informed that the product must be rebuilt so that high levels of quality can be maintained, stakeholders cry foul and demand that **"a few fixes" be applied to make the prototype a working product**. Too often, software development management relents.`

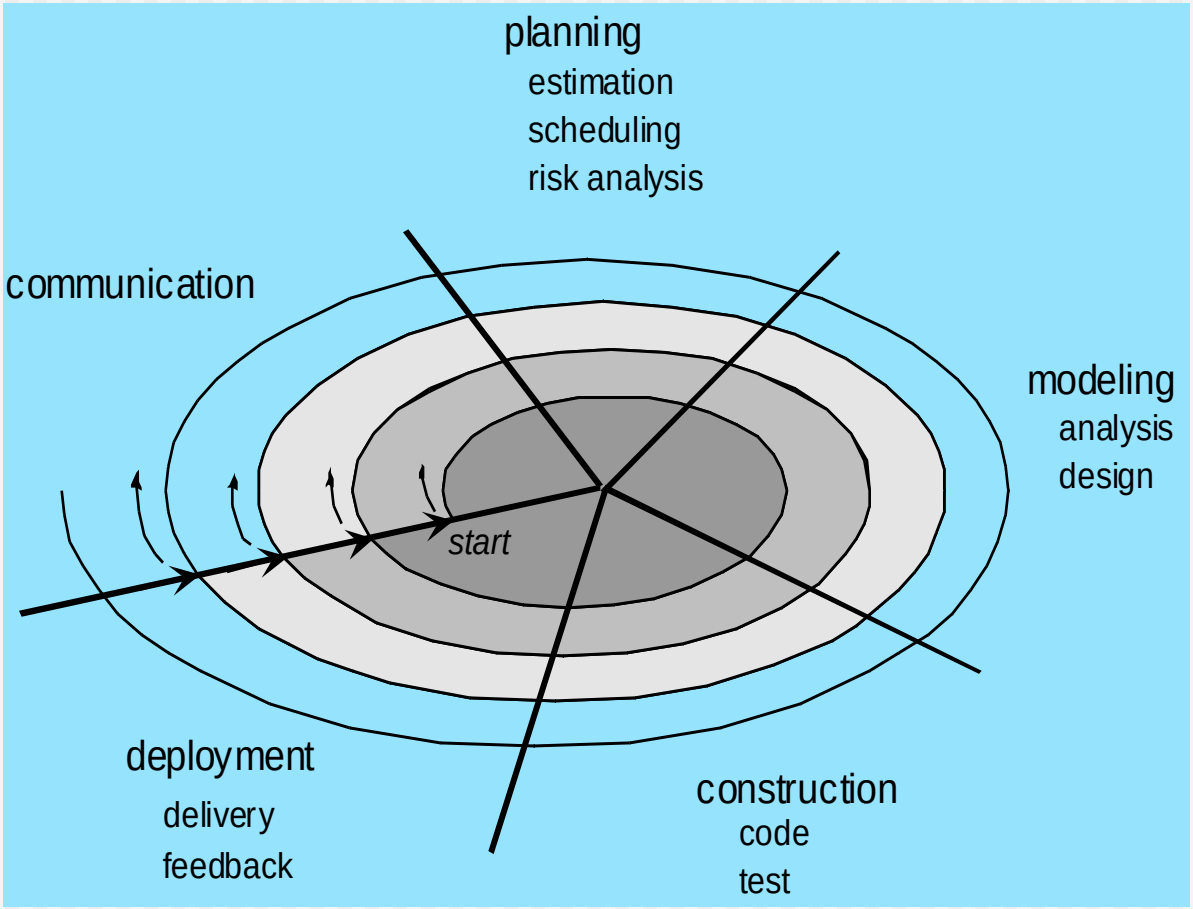
# Evolutionary Models: Prototyping

- **2.** A software engineer, often compromises implementation in order to get a prototype working quickly.
  - An **inappropriate operating system or programming language** may be used simply **because it is available and known**;
  - an **inefficient algorithm** may be implemented **simply to demonstrate capability.**
- After a time, you may become comfortable with these choices and forget all the reasons why they were inappropriate.
- The less-than-ideal choice has now become an integral part of the system.

# Evolutionary Models: Prototyping

- Although problems can occur, prototyping can be an effective paradigm for software engineering.
- The **key** is to define the rules of the game at the beginning; that is, **all stakeholders should agree that the prototype is built to serve as a mechanism for defining requirements**.
- It is **then discarded** (at least in part), and the **actual software is engineered with an eye toward quality.**
  - **An early example of large-scale software prototyping was the implementation of NYU's Ada/ED translator for the Ada programming language.[2] It was implemented in SETL with the intent of producing an executable semantic model for the Ada language, emphasizing clarity of design and user interface over speed and efficiency. The NYU Ada/ED system was the first validated Ada implementation, certified on April 11, 1983.**

# Evolutionary Models: The Spiral



# Evolutionary Models: The Spiral

- This is proposed by Barry Boehm.
- It couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model and is a risk-driven process model generator that is used to guide multi-stakeholder concurrent engineering of software intensive systems.
- Two main distinguishing features:
  - One is cyclic approach for incrementally growing a system's degree of definition and implementation while decreasing its degree of risk.
  - The other is a set of anchor point milestones for ensuring stakeholder commitment to feasible and mutually satisfactory system solutions.

# Evolutionary Models: The Spiral

- **A series of evolutionary releases are delivered.**
- **During the early iterations, the release might be a *model or prototype.***
- **During later iterations, increasingly more *complete version of the engineered system are produced.***
- A spiral model is divided into a set of framework activities defined by the software engineering team.
- **Each** of the framework activities represent **one segment** of the spiral path illustrated in Figure above.
- As this evolutionary process begins, the software team performs activities that are implied by a circuit around the spiral in a clockwise direction, beginning at the center.

# Evolutionary Models: The Spiral

- **Risk** is considered as each revolution is made.
- ***Anchor point milestones***—a combination of work products and conditions that are attained along the path of the spiral—are noted for each evolutionary pass.
- The first circuit around the spiral might result in the development of a **product specification**; subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software.
  - **Each pass through the planning region results in adjustments to the project plan.**
  - **Cost and schedule are adjusted based on feedback derived from the customer after delivery.**
  - **The project manager adjusts the planned number of iterations required to complete the software.**

# Evolutionary Models: The Spiral

- Unlike other process models that end when software is delivered, the spiral model can be adapted to apply throughout the life of the computer software.
- Therefore, the first circuit around the spiral might represent a **"concept development project"** that starts at the core of the spiral and continues for multiple iterations until concept development is complete.
- If the concept is to be developed into an actual product, the process proceeds outward on the spiral and a **"new product development project"** commences.
- The new product will evolve through a number of iterations around the spiral.
- Later, a circuit around the spiral might be used to represent a **"product enhancement project."**

# Evolutionary Models: The Spiral

- The spiral, when characterized in this way, remains operative until the software is retired. There are times when the process is dormant, but whenever a change is initiated, the process starts at the appropriate entry point (e.g., product enhancement).
- The spiral model is a realistic approach to the development of large-scale systems and software.
- Because software evolves as the process progresses, the **developer and customer better understand and react to risks at each evolutionary level**.
- The spiral model uses prototyping as a risk reduction mechanism and enables to apply the prototyping approach at any stage in the evolution of the product.

# Evolutionary Models: The Spiral

- It maintains the systematic stepwise approach suggested by the classic life cycle but incorporates it into an iterative framework that more realistically reflects the real world.
- The spiral model demands a direct consideration of technical risks at all stages of the project and, if properly applied, should reduce risks before they become problematic.
- **Disadvantages:**
- But like other paradigms, the spiral model is not a **panacea**.
- It may be **difficult to convince customers** (particularly in contract situations) that the evolutionary approach is controllable.
- It **demands** considerable **risk assessment expertise** and relies on this expertise for success.
- If a major **risk** is **not uncovered** and managed, **problems** will undoubtedly **occur**.

# Unified Process

- Unified Process—a **"use-case driven, architecture-centric, iterative and incremental"** software process closely aligned with the Unified Modeling Language (UML) proposed by **Ivar Jacobson, Grady Booch, and James Rumbaugh .**
- The Unified Process recognizes the **importance of customer communication and streamlined methods** for describing the **customer's view of a system (the use case).**
- It emphasizes the important role of software architecture and "helps the architect focus on the right **goals**, such as **understandability, reliance to future changes, and reuse".**

# Unified Process contd.,

- **Unified Process Brief History:** During the early 1990s **James Rumbaugh, Grady Booch, and Ivar Jacobson** began working on a **"unified method"** that would combine the best features of each of their individual object-oriented analysis and design methods and adopt additional features proposed by other experts in object-oriented modelling.
- The result was **UML**—a *unified modelling language* that **contains a robust notation for the modelling and development of object-oriented systems. By 1997, UML became a de facto industry standard for object-oriented software development**

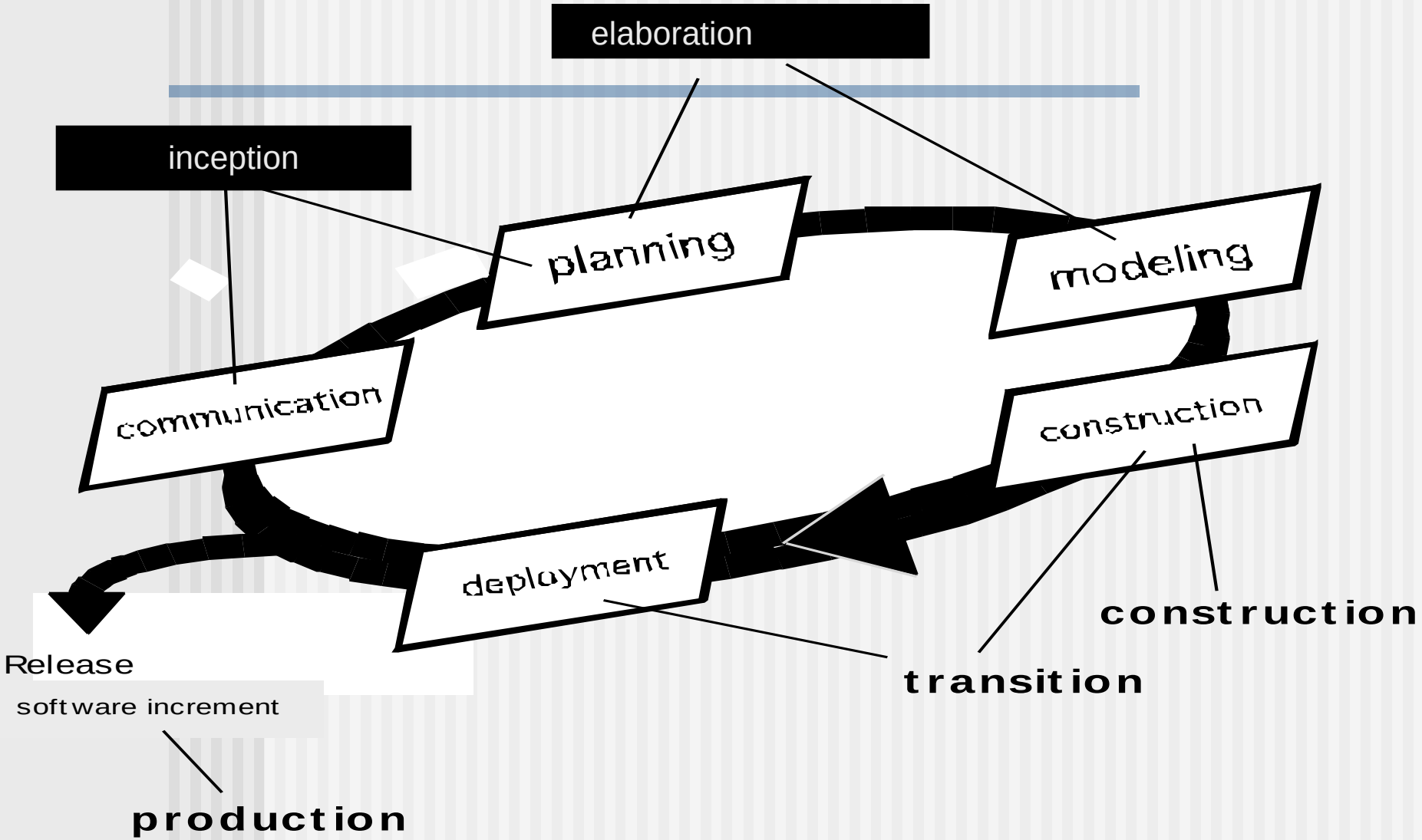
# Unified Process contd.,

- UML, is used to represent both requirements and design models, presents an introductory tutorial for those who are unfamiliar with basic UML notation and modelling rules.
- **UML provided** the necessary **technology** to support object-oriented software engineering practice, but it **did not provide** the **process framework** to guide project teams in their application of the technology.
- Over the next few years, **Jacobson, Rumbaugh, and Booch developed the *Unified Process,* a framework for object-oriented software engineering using UML**.

# Unified Process contd.,

- Today, the Unified Process (UP) and UML are widely used on object-oriented projects of all kinds.
- **The iterative, incremental model proposed by the UP can and should be adapted to meet specific project needs.**
- Phases of the Unified Process:
- **Five generic framework activities are used to describe any software process model. The Unified Process also use these same activities.**
- Figure (next slide) depicts the “phases” of the UP and relates them to the generic activities

# The Unified Process (UP)



# Unified Process contd.,

- The ***inception phase*** of the UP encompasses both **customer communication and planning activities.**
- By collaborating with stakeholders,
  - **business requirements** for the software are identified;
  - a **rough architecture** for the system is proposed; and
  - a **plan** for the iterative, incremental nature of the ensuing project is developed.
- **Fundamental business requirements** are described through a **set of preliminary use cases** that **describe which features and functions** each major class of users desires.

# Unified Process contd.,

- **Architecture,** here, is **a tentative outline of major subsystems and the function and features that populate them.**
- Later, the **architecture will be refined and expanded** into a **set of models** that will represent different views of the system.
- **Planning identifies resources, assesses major risks, defines a schedule, and establishes a basis** for the phases that are to be applied as the software increment is developed.

# Unified Process contd.,

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

# Unified Process contd.,

- In some cases, elaboration creates an **“executable architectural baseline”** that represents a **“first cut”** executable system.
- The architectural baseline demonstrates the viability of the architecture but **does not provide all features and functions** required to use the system.
- The **plan** is carefully **reviewed at the culmination of the elaboration phase to ensure that scope, risks, and delivery dates remain reasonable.**
- Modifications to the plan are often made at this time.

# Unified Process contd.,

- The ***construction phase*** of the UP is identical to the construction activity defined for the generic software process.
- Using the **architectural model as input**, the construction phase **develops** or acquires the **software components** that will make **each use case operational for end users.**
- To accomplish this, **requirements and design models that were started during the elaboration phase are completed to reflect the final version of the software increment.**

# Unified Process contd.,

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

# Unified Process contd.,

- The ***transition phase*** of the UP encompasses the latter stages of the **generic construction activity and the first part of the generic deployment (delivery and feedback) activity.**
- **Software is given to end users for beta testing and user feedback reports both defects and necessary changes**.
- The software team creates the necessary support information (e.g., **user manuals, troubleshooting guides, installation procedures**) that is required for the release.
- At the conclusion of the transition phase, the software increment becomes a **usable software release**.

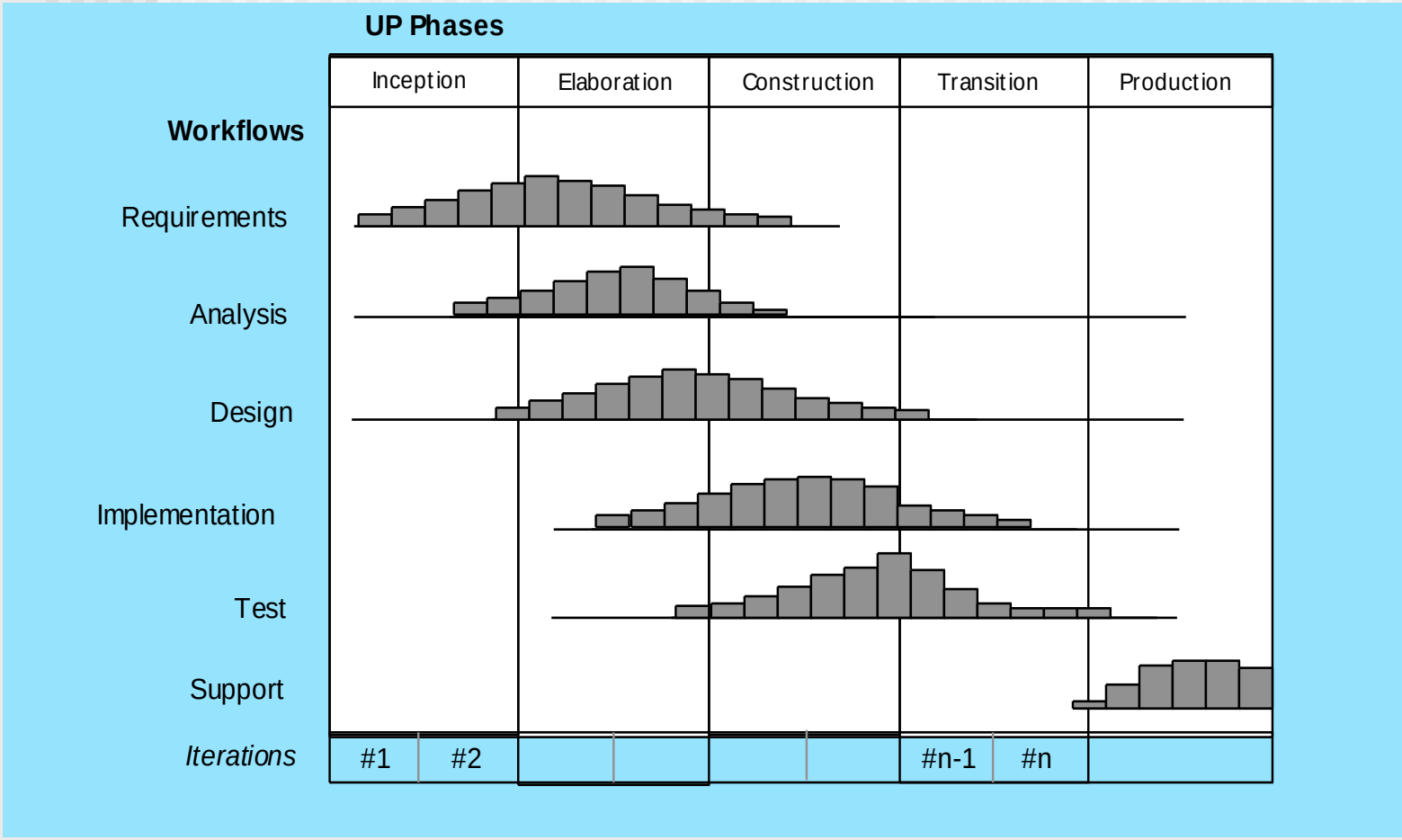
# Unified Process contd.,

- The ***production phase*** of the UP coincides with the **deployment activity** of the generic process.
- During this phase, the **on going use of the software is monitored, support for the operating environment (infrastructure) is provided, and defect reports and requests for changes are submitted and evaluated**.
- It is likely that at the same time the construction, transition, and production phases are being conducted, work may have already begun on the next software increment.
- This means that the five UP phases do not occur in a sequence, but rather with staggered concurrency.

# Unified Process contd.,

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

# UP Phases


UP Phases
Inception
Elaboration
Construction
Transition
Production
Workflows
Requirements
Analysis
Design
Implementation
Test
Support
Iterations
#1
#2
#n-1
#n

# UP Work Products

## Inception phase

- Vision document
- Initial use-case model
- Initial project glossary
- Initial business case
- Initial risk assessment.
- Project plan,
  - phases and iterations.
- Business model,
  - if necessary.
- One or more prototypes

## Elaboration phase

- Use-case model
- Supplementary requirements
  - including non-functional
- Analysis model
- Software architecture
  - Description.
- Executable architectural
  - prototype.
- Preliminary design model
- Revised risk list
- Project plan including
  - iteration plan
  - adapted workflows
  - milestones
  - technical work products
- Preliminary user manual

## Construction phase

- Design model
- Software components
- Integrated software
  - increment
- Test plan and procedure
- Test cases
- Support documentation
  - user manuals
  - installation manuals
  - description of current
    - increment

## Transition phase

- Delivered software increment
- Beta test reports
- General user feedback

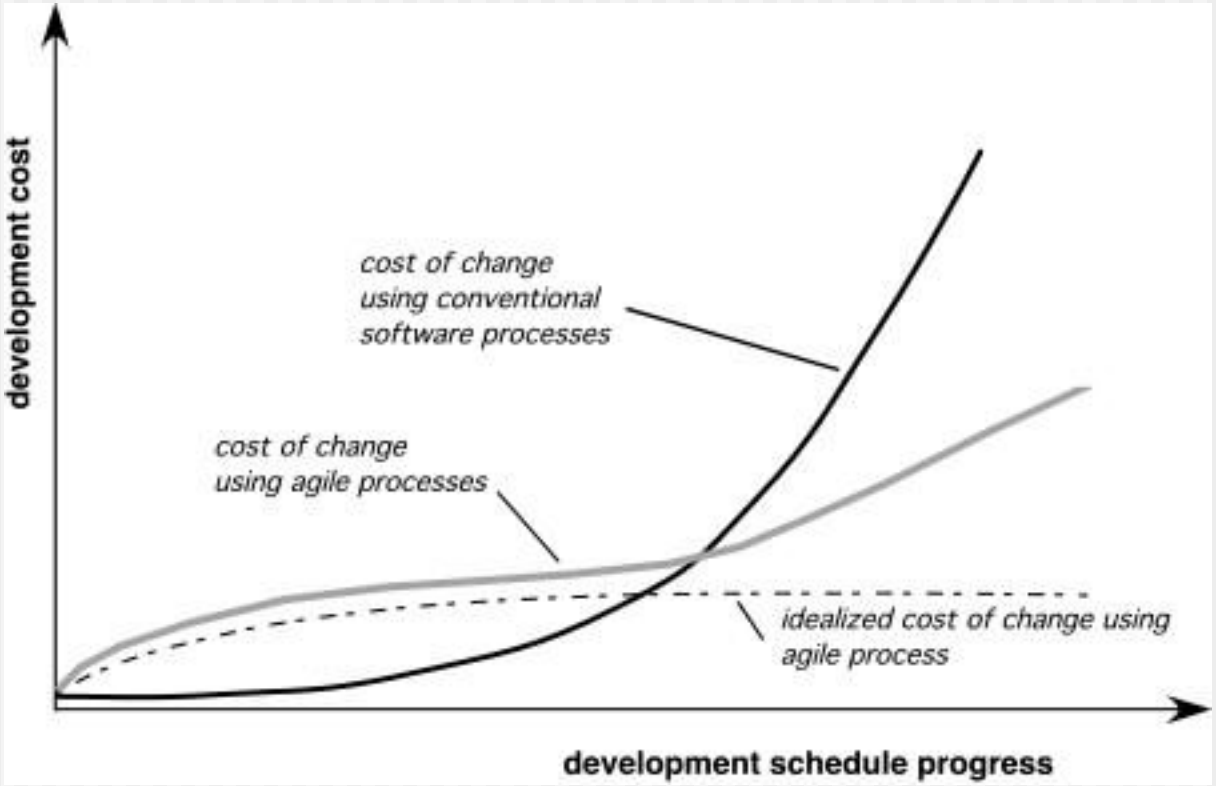
# What is “Agility”?

- Effective (rapid and adaptive) response to change
- Effective communication among all stakeholders
- Drawing the customer onto the team
- Organizing a team so that it is in control of the work performed

*Yielding …*

- Rapid, incremental delivery of software

# Agility and the Cost of Change



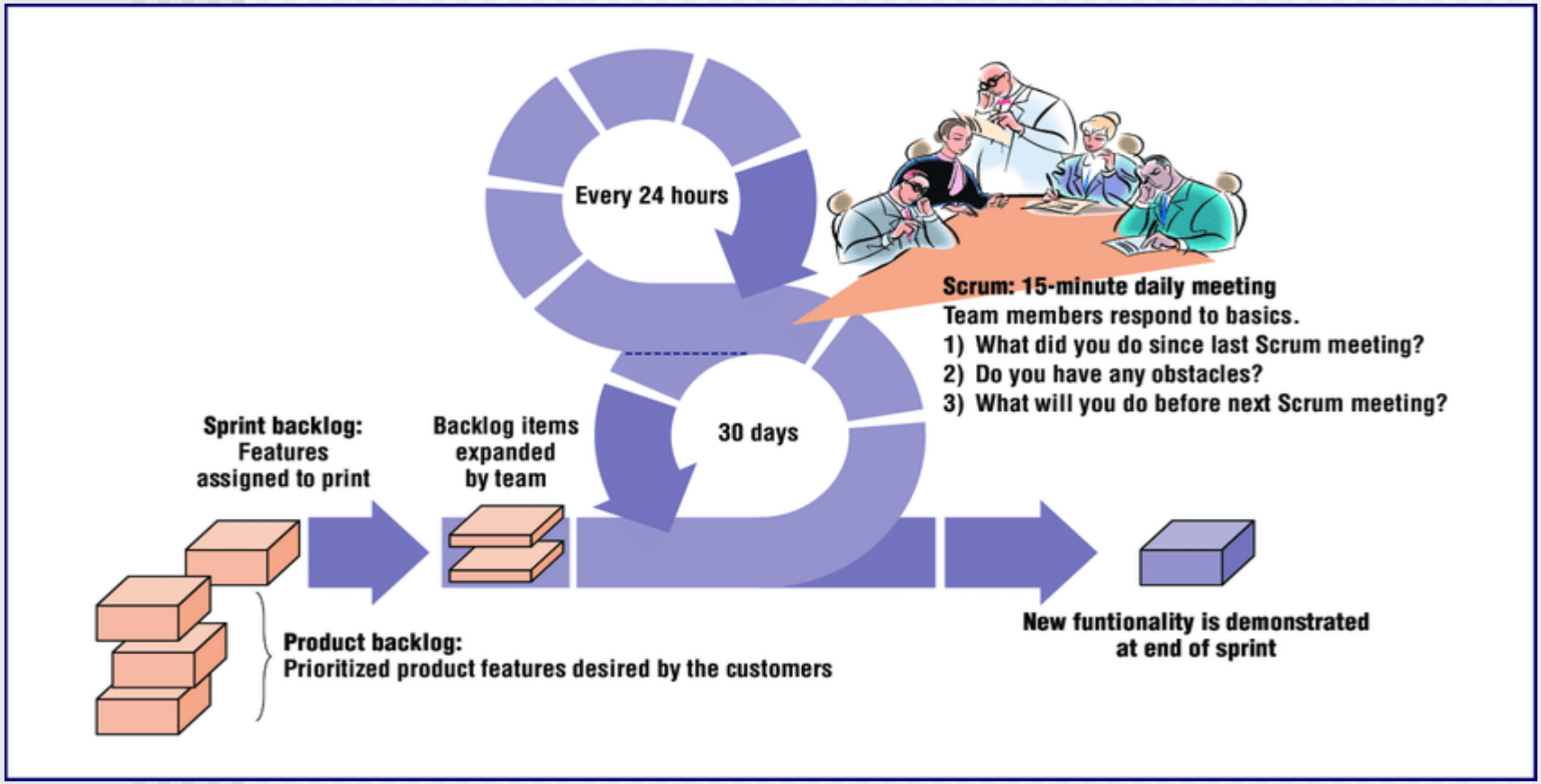
# An Agile Process

- Is driven by customer descriptions of what is required (scenarios)
- Recognizes that plans are short-lived
- Develops software iteratively with a heavy emphasis on construction activities
- Delivers multiple ‘software increments’
- Adapts as changes occur

# Scrum

- Originally proposed by Schwaber and Beedle
- Scrum—distinguishing features
  - Development work is partitioned into "packets"
  - Testing and documentation are on-going as the product is constructed
  - Work occurs in "sprints" and is derived from a "backlog" of existing requirements
  - Meetings are very short and sometimes conducted without chairs
  - "demos" are delivered to the customer with the time-box allocated

# Scrum


Every 24 hours
Scrum: 15-minute daily meeting
Team members respond to basics.
1) What did you do since last Scrum meeting?
2) Do you have any obstacles?
3) What will you do before next Scrum meeting?
30 days
Sprint backlog:
Features
assigned to print
Backlog items
expanded
by team
Product backlog:
Prioritized product features desired by the customers
New funtionality is demonstrated
at end of sprint