Defining 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 inform and

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

1. The Nature of Software

Today, software takes on a dual role. *It is a* ***product****, and at the same time, the* ***vehicle*** *for delivering a product.*

As a product*,* it delivers the computing potential embodied by computer hardware or more broadly, *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.***

As the vehicle used to deliver the product, software acts as the basis for the control of the computer (operating systems), the communication of information (networks), and the creation and control of other programs (software tools and environments).

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.

Software manages business information to enhance competitiveness;
Software provides a gateway to worldwide information networks (e.g., the Internet).
Software provides the means for acquiring information in all of its forms.
Software role has undergone significant change over the last half-century.
Software industry has become a dominant factor industrialized world.

Legacy Software

Older programs —often referred to as *legacy software*—have been the focus of continuous attention and concern since the 1960s. Dayani-Fard and his colleagues [Day99] describe legacy software in the following way:

> Legacy software systems . . . were developed decades ago and have been continually modified to meet changes in business requirements and computing platforms. The maintenance of such systems is causing headaches for large organizations who find them costly to maintain and risky to evolve.

Liu and his colleagues [Liu98] extend this description by noting that "many legacy systems remain supportive to core business functions and are 'indispensable' to the business." Hence, legacy software is characterized by longevity and business criticality.

Unfortunately, there is sometimes one additional characteristic that is present in legacy software—*poor quality.*
Legacy systems sometimes have inextensible designs, convoluted code, poor or nonexistent documentation, test cases and results that were never archived, a poorly managed change history—the list can be quite long.

The only reasonable answer may be: *Do nothing,* at least until the legacy system must undergo some significant change. If the legacy software meets the needs of its users and runs reliably, it isn't broken and does not need to be fixed. However, as time passes, legacy systems often evolve for one or more of the following reasons:

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

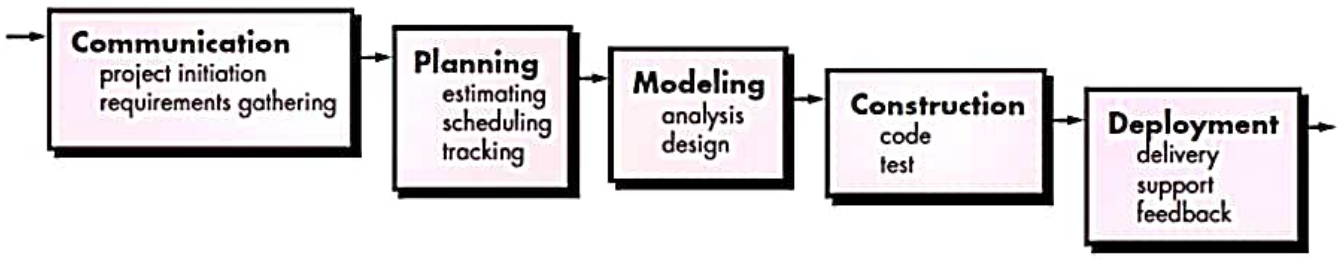
When these modes of evolution occur, a legacy system must be reengineered, so that it remains useful in the future. The goal of modern software engineering is to "devise methodologies that are founded on the notion of evolution"; that is, the notion that software systems continually change, new software systems are built from the old ones, and . . . all must interoperate and cooperate with each other".

The Waterfall Model

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

Context: Used when requirements are reasonably well understood.

Advantage: It can serve as a useful process model in situations where requirements are fixed and work is to proceed to complete in a linear manner.



The **problems** that are sometimes encountered when the ***waterfall model*** is applied are:

i. 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.

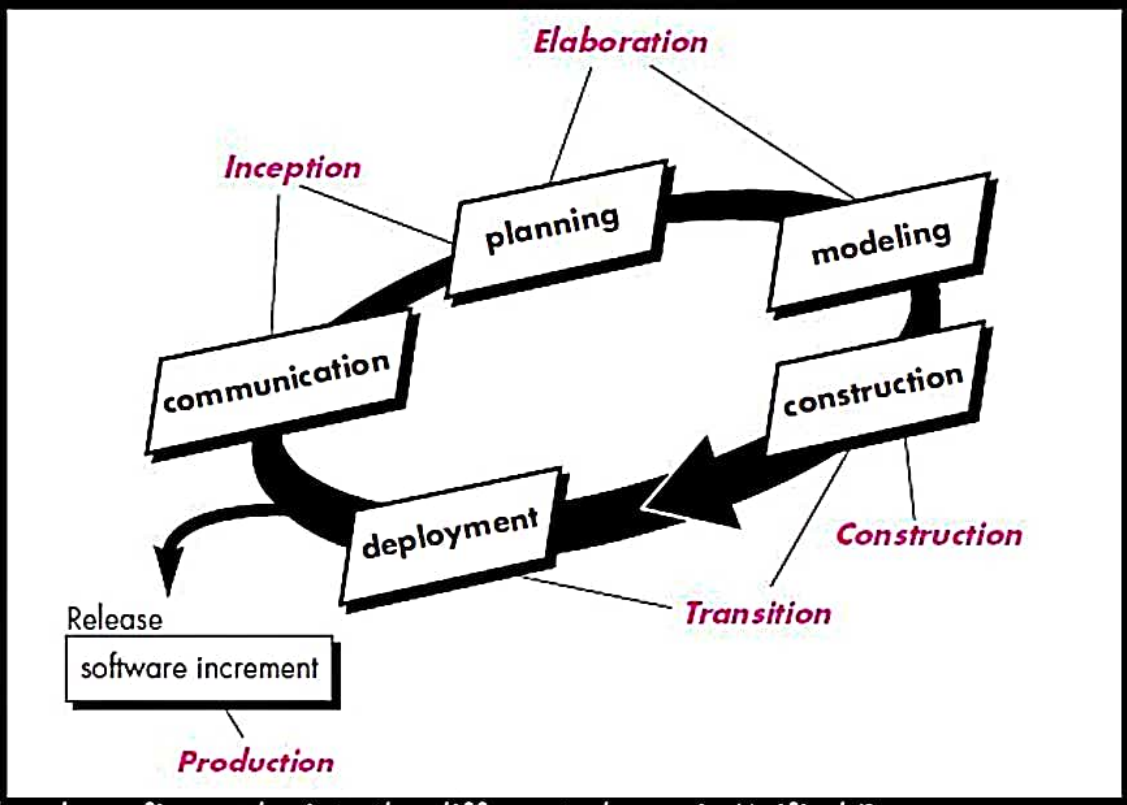
ii. It is often difficult for the customer to state all requirements explicitly. The waterfall model requires this and has difficulty accommodating the natural uncertainty that exist at the beginning of many projects.

iii. The customer must have patience. A working version of the programs will not be available until late in the project time-span. If a major blunder is undetected then it can be disastrous until the program is reviewed.

11. THE UNIFIED PROCESS

- The unified process related to "use case driven, architecture-centric, iterative and incremental" software process.
- The Unified Process is an attempt to draw on the best features and characteristics of traditional software process models.
- The Unified Process recognizes the importance of customer communication and streamlined methods for describing the customer's view of a system.
- It emphasizes the important role of software architecture and "helps the architect focus on the right goals.
- It suggests a process flow that is iterative and incremental.
- During the early 1990s James Rumbaugh [Rum91], Grady Booch [Boo94], and Iva Jacobson [Jac92] began working on a "unified method".
- The result was UML—*a unified modeling language* that contains a robust notation for the modeling and development of object-oriented systems.
- By 1997, UML became a de facto industry standard for object-oriented software development.
- UML is used to represent both requirements and design models.
- UML provided the necessary technology to support object-oriented software engineering practice, but it did not provide the process framework.
- Over the next few years, Jacobson, Rumbaugh, and Booch developed the *Unified Process*, a framework for object-oriented software engineering using UML.
- Today, the Unified Process (UP) and UML are widely used on object-oriented projects of all kinds.
- The iterative, incremental model proposed by the UP can and should be adapted to meet specific project needs.

Phases of the Unified Process



- The above figure depicts the different phases in Unified Process.
- The ***inception phase*** of the UP encompasses both customer communication and planning activities.
  - o By collaborating with stakeholders, business requirements for the software are identified;
  - o a rough architecture for the system is proposed; and
  - o a plan for the iterative, incremental nature of the ensuing project is developed. o Fundamental business requirements are described.
  - o The architecture will be refined.
  - o Planning identifies resources, assesses major risks, defines a schedule, and establishes a basis for the phases.
- The **elaboration phase** encompasses the communication and modeling activities of the generic process model.
  - o 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.
  - o In some cases, elaboration creates an "executable architectural baseline" that represents a "first cut" executable system.
  - o The architectural baseline demonstrates the viability of the architecture but does not provide a d functions required to use the syster
  - o In addition, the plan is ewed.
  - o Modifications to the plan are often made at this time.
- The ***construction phase*** of the UP is identical to the construction activity define for the generic software process. Using the architectural model as input, th construction phase develops or acquires the software components that will mal each use case operational for end users.
  - o The elaboration phase reflect the final version of the software increment.

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

- The ***transition phase*** of the UP encompasses the latter stages of the generic construction activity and the first part of the generic deployment (delivery and feedback) activity.
  - Software is given to end users for beta testing and user feedback reports both defects and necessary changes.
  - In addition, the software team creates the necessary support information (e.g., user manuals, troubleshooting guides, installation procedures) that is required for the release.
  - At the conclusion of the transition phase, the software increment becomes a usable software release.
- The ***production phase*** of the UP coincides with the deployment activity of the generic process.
  - During this phase, the ongoing use of the software is monitored, support for the operating environment (infrastructure) is provided, and defect reports and requests for changes are submitted and evaluated.
  - It is likely that at the same time the construction, transition, and production phases are being conducted.
  - Work may have already begun on the next software increment.
  - This means that the five UP phases do not occur in a sequence, but rather with staggered concurrency.

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

4. 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) 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.
- An *action* (e.g., architectural design) 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.

In the context of software engineering, a process is *not* a rigid prescription for how to build computer software. Rather, it 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.

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.

A generic process framework for software engineering encompasses five activities:

i. **Communication.** Before any technical work can commence, it is critically important to communicate and collaborate with the customer (and other stakeholders11 The intent is to understand stakeholders' objectives for the project and to gather requirements that help define software features and functions.
ii. **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.
iii. **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—what it will look like architecturally, how the constituent parts fit together, and many other characteristics. 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.
iv. **Construction.** This activity combines code generation (either manual or automated) and the testing that is required to uncover errors in the code.
v. **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.

For many software projects, framework activities are applied iteratively as a project progresses. That is,

communication,
planning,
modeling,
construction, and
deployment are applied repeatedly through a number of project iterations. Each project iteration produces a *software increment* that provides stakeholders with a subset of overall software features and functionality. As each increment is produced, the software becomes more and more complete.

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.

iv. **Construction.** This activity combines code generation (either manual or automated) and the testing that is required to uncover errors in the code.
v. **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.

For many software projects, framework activities are applied iteratively as a project progresses. That is,

communication,
planning,
modeling,
construction, and
deployment are applied repeatedly through a number of project iterations. Each project iteration produces a *software increment* that provides stakeholders with a subset of overall software features and functionality. As each increment is produced, the software becomes more and more complete.

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 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.
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. Each of these umbrella activities is discussed in detail later in this book.

The software engineering process is not a rigid prescription that must be followed dogmatically by a software team. Rather, 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

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

6. Software Myths

- Software myths—erroneous beliefs about software and the process that is used to build it—can be traced to the earliest days of computing.
- Myths have a number of attributes that make them insidious. For instance, they appear to be reasonable statements of fact (sometimes containing elements of truth), they have an intuitive feel, and they are often announced by experienced practitioners who "know the score."
- Today, most knowledgeable software engineering professionals recognize myths for what they are—misleading attitudes that have caused serious problems for managers and practitioners alike.
- However, old attitudes and habits are difficult to modify, and remnants of software myths remain.
- Software Myths are three types 1. Managers Myths
  2. Customers Myths (and other non-technical stakeholders)
  3. Practitioners Myths

i. **Management myths.** Managers with software responsibility, like managers in most disciplines, are often under pressure to maintain budgets, keep schedules from slipping, and improve quality. Like a drowning person who grasps at a straw, a software manager often grasps at belief in a software myth, if that belief will lessen the pressure (even temporarily).

Myth: *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: The book of standards may very well exist, but is it used?
Are software practitioners aware of its existence?
Does it reflect modern software engineering practice?
Is it complete?
Is it adaptable?
Is it streamlined to improve time-to-delivery while still maintaining a focus on quality?

In many cases, the answer to all of these questions is "no."

Myth: *If we get behind schedule, we can add more programmers and catch up.*
Reality: Software development is not a mechanistic process like manufacturing. In the words of Mr. Brooks "adding people to a late software project makes it later."
At first, this statement may seem counterintuitive. However, as new people are added, people who were working must spend time educating the newcomers, thereby reducing the amount of time spent on productive development effort.

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

Myth: *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 outsources software projects.

ii. **Customer myths.** A customer who requests computer software may be a person at the next desk, a technical group down the hall, the marketing/sales department, or an outside company that has requested software under contract. In many cases, the customer believes myths about software because software managers and practitioners do little to correct misinformation. Myths lead to false expectations (by the customer) and, ultimately, dissatisfaction with the developer.

Myth: *A general statement of objectives is sufficient to begin writing programs—we can fill in the details later.*
Reality: Although a comprehensive and stable statement of requirements is not always possible, an ambiguous "statement of objectives" is a recipe for disaster. Unambiguous requirements (usually derived iteratively) are developed only through effective and Continuous communication between customer and developer.

Myth: *Software requirements continually change, but change can be easily accommodated because software is flexible.*
Reality: It is true that software requirements change, but the impact of change varies with the time at which it is introduced. When requirements changes are requested early (before design or code has been started), the cost impact is relatively small. However, as time passes, the cost impact grows rapidly—resources have been committed, a design framework has been established, and change can requires additional resources and major design modification.

iii. **Practitioner's myths.** Myths that are still believed by software practitioners have been fostered by over 50 years of programming culture. During the early days, programming was viewed as an art form. Old ways and attitudes die hard.

Myth: *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.

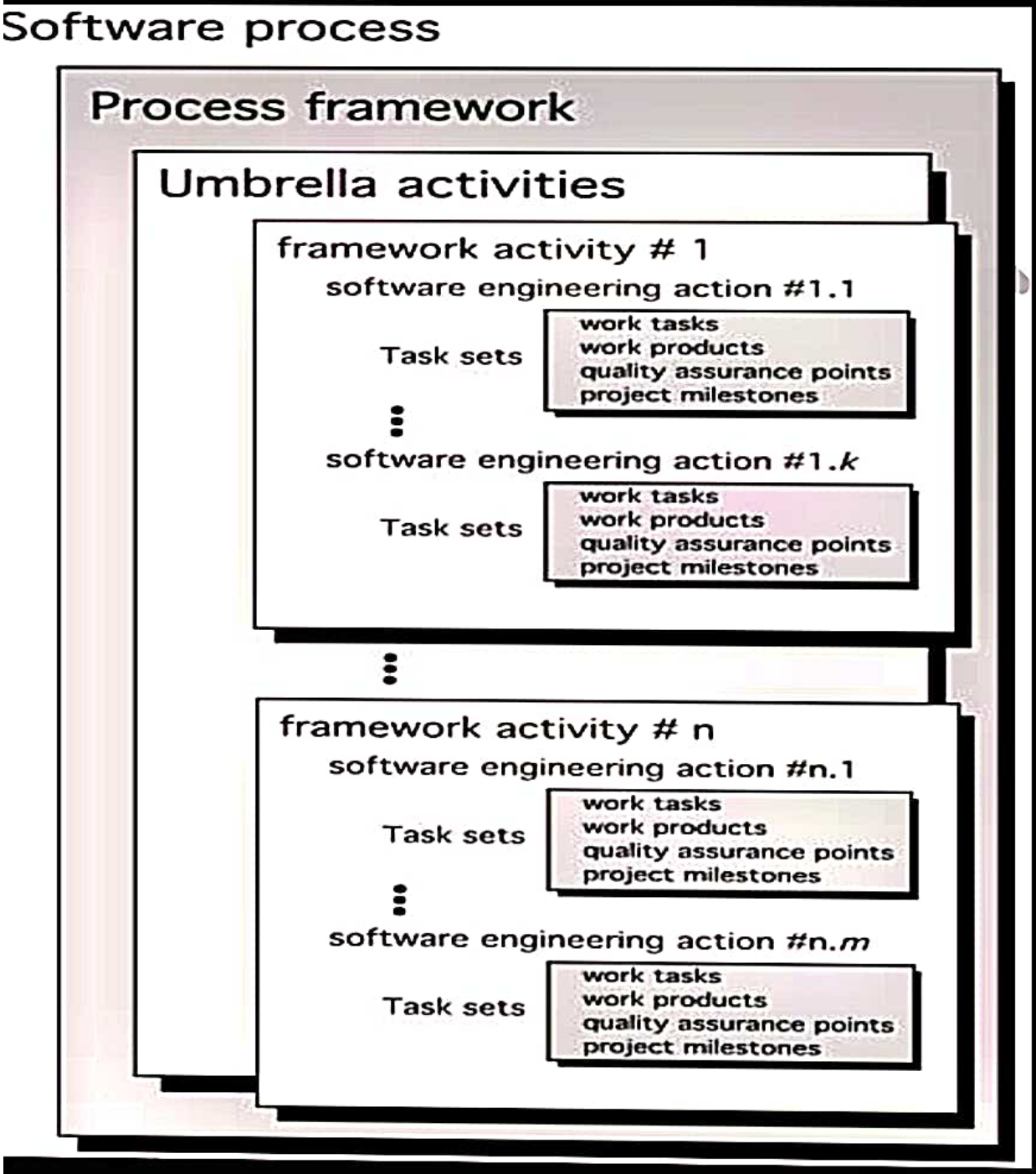
Myth: *Until I get the program "running" I have no way of assessing its quality.*
Reality: One of the most effective software quality assurance mechanisms can be applied from the inception of a project—*the technical review.* Software reviews are a "quality filter" that have been found to be more effective than testing for finding certain classes of software defects.

Myth: *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. A variety of work products (e.g., models, documents, plans) provide a foundation for successful engineering and, more important, guidance for software support.

Myth: *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 a quality product. Better quality leads to reduced rework. And reduced rework results in faster delivery times. Many software engineers have adopted an "agile" approach that accommodates change incrementally, thereby controlling its impact and cost.

7. A 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.



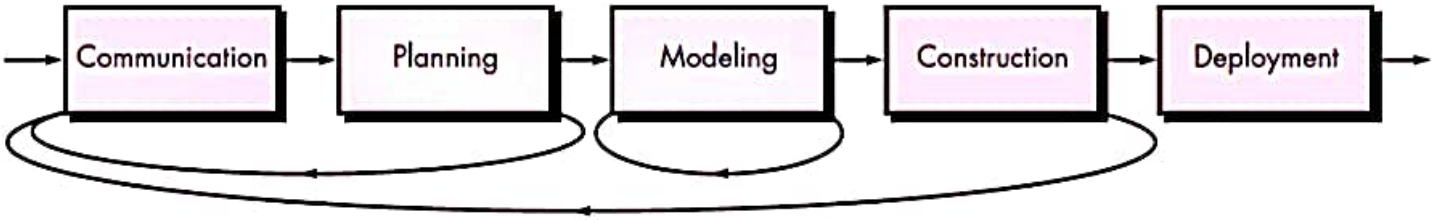
- The software process is represented schematically in the above figure.
- Referring to the figure, each framework activity is populated by a set of software engineering actions.
- Each software engineering action is defined by a *task set* that identifies o the work tasks that are to be completed,
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

- A generic process framework for software engineering defines five framework activities
  - o **communication,**
  - o **planning,**
  - o **modeling,**
  - o **construction,** and
  - o **deployment.**
- In addition, a set of umbrella activities—project tracking and control, risk management, quality assurance, configuration management, technical reviews, and others—are applied throughout the process.

- The *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 and is illustrated in the above Figure.
- A linear *process flow* executes each of the five framework activities in sequence, beginning with communication and culminating with deployment which is shown in the following Figure.

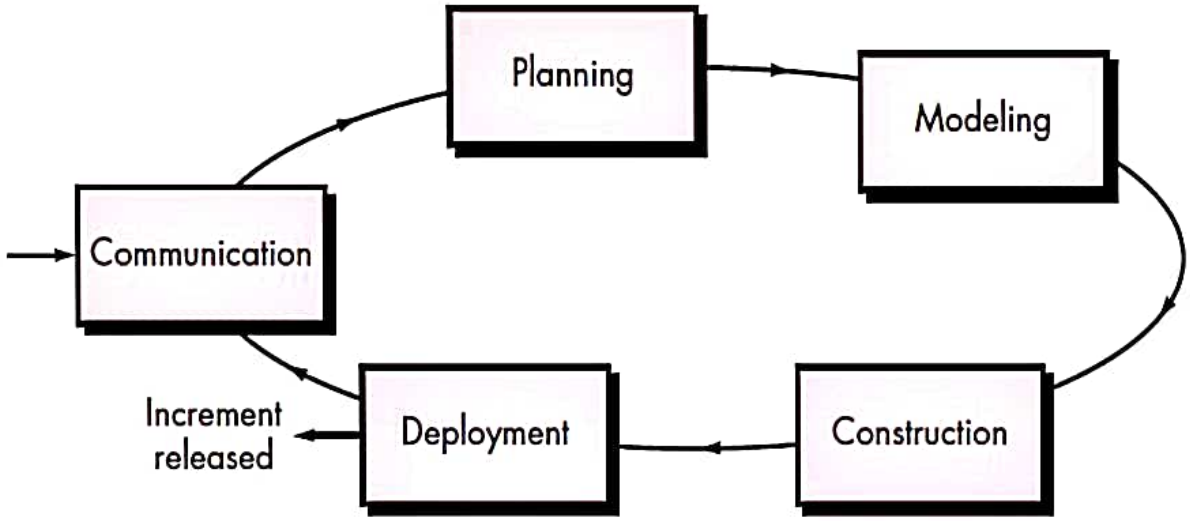
(a) Linear process flow

- An *iterative process flow* repeats one or more of the activities before proceeding to the next (shown in the following Figure).



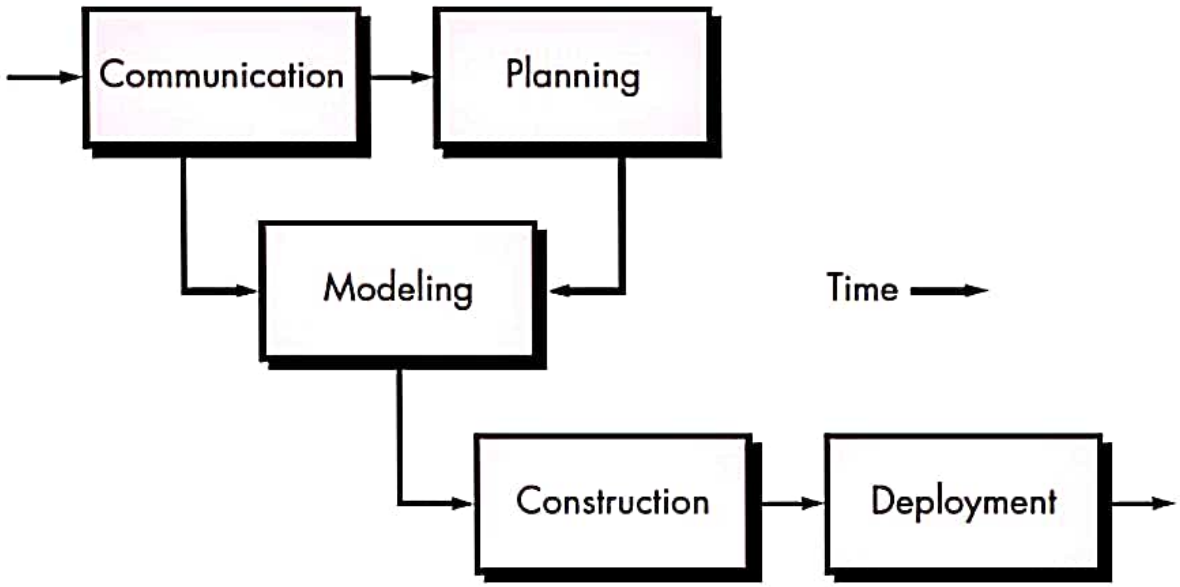
(b) Iterative process flow

- An *evolutionary process flow* executes the activities in a "circular" manne
  Each circuit through the five activities leads to a more complete version o
  the software (shown in the following Figure).



(c) Evolutionary process flow

- A *parallel process flow* executes one or more activities in parallel with
  other activities
  e.g., modeling for one aspect of the software might be executed in paralle
  with construction of another aspect of the software. (shown in th
  following figure)



(d) Parallel process flow

Defining a Framework Activity

There are five framework activities, they are

- communication,
- planning,
- modeling,
- construction, and
- deployment.

These five framework activities provide a basic definition of Software Process. These Framework activities provides basic information like *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?*

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.

If the project was considerably more complex with many stakeholders, each with a different set of requirements, the communication activity might have six distinct actions: *inception, elicitation, elaboration, negotiation, specification,* and *validation.* Each of these software engineering actions would have many work tasks and a number of distinct work products.

Process Patterns

- Every software team encounters problems as it moves through the software process.
- It would be useful if proven solutions to these problems were readily available to the team so that the problems could be addressed and resolved quickly.
- A *process pattern*1 describes a process-related problem that is encountered during software engineering work, identifies the environment in which the problem has been encountered, and suggests one or more proven solutions to the problem.
- Stated in more general terms, a process pattern provides you with a template a consistent method for describing problem solutions within the context of the software process.
- By combining patterns, a software team can solve problems and construct a process that best meets the needs of a project.
- Patterns can be defined at any level of abstraction.
- In some cases, a pattern might be used to describe a problem and solution associated with a complete process model (e.g., prototyping).

- In other situations, patterns can be used to describe a problem and solution associated with a framework activity (e.g., **planning**) or an action within a framework activity (e.g., project estimating).
- Ambler has proposed a template for describing a process pattern:

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

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

Type. The pattern type is specified. Ambler suggests three types:

1. ***Stage pattern***—defines a problem associated with a framework activity for the process. Since a framework activity encompasses multiple actions and work tasks, a stage pattern incorporates multiple task patterns (see the following) that are relevant to the stage (framework activity).

An example of a stage pattern might be **EstablishingCommunication**. This pattern would incorporate the task pattern **RequirementsGathering** and others.

2. ***Task pattern***—defines a problem associated with a software engineering action or work task and relevant to successful software engineering practice (e.g., **RequirementsGathering** is a task pattern).

3. ***Phase pattern***—define the sequence of framework activities that occurs within the process, even when the overall flow of activities is iterative in nature. An example of a phase pattern might be **SpiralModel** or **Prototyping**.

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 known.

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

Solution. Describes how to implement the pattern successfully. This section 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.

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?

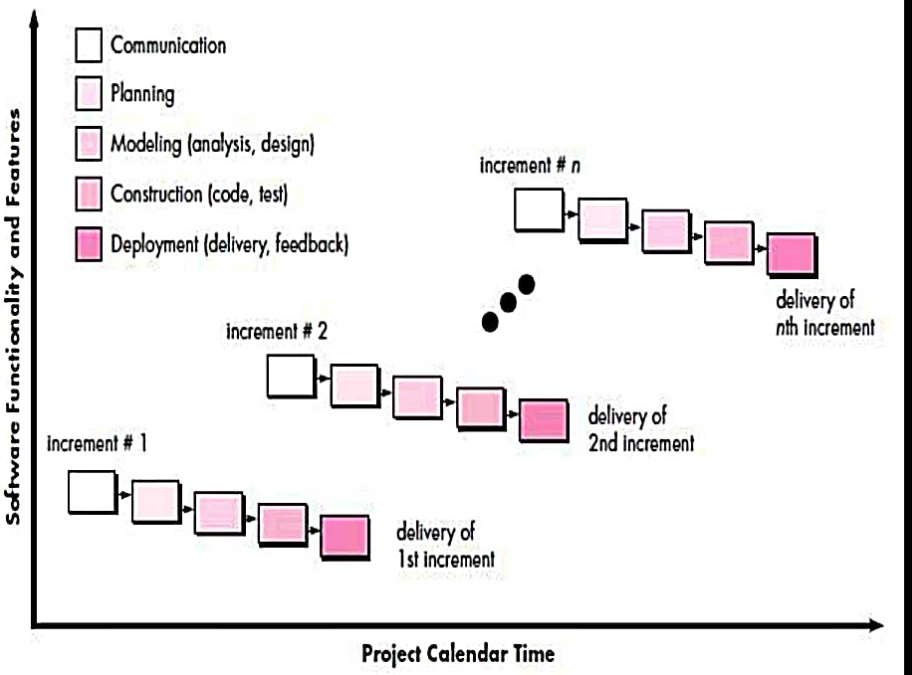
Related Patterns. Provide a list of all process patterns that are directly related to this one. This may be represented as a hierarchy or in some other diagrammatic form. For example, the stage pattern **Communication** encompasses the task patterns: **ProjectTeam**, **CollaborativeGuidelines**, **ScopeIsolation**, **RequirementsGathering**, **ConstraintDescription**, and **ScenarioCreation**.

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.

Conclusion on Process Patterns

- Process patterns provide an effective mechanism for addressing problems associated with any software process.
- The patterns enable you to develop a hierarchical process description that begins at a high level of abstraction (a phase pattern).
- The description is then refined into a set of stage patterns that describe framework activities
- Once process patterns have been developed, they can be reused for the definition of process variants..

Incremental Process Models



- There are many situations in which initial software requirements are reasonably well defined, but the overall scope of the development effort difficulty to implement linear process.
- Need to provide a limited set of software functionality to users quickly and then refine and expand on that functionality in later software releases.
- In such cases, you can choose a process model that is designed to produce the software in increments.
- The *incremental* model combines elements of linear and parallel process flows. The above Figure shows the incremental model which applies linear sequences
- Each linear sequence produces deliverable "increments" of the software in a manner that is similar to the increments produced by an evolutionary process flow.
- For example, MS-Word software developed using the incremental paradigm might deliver
    - o basic file management, editing, and document production functions in the first increment;
    - o more sophisticated editing and document production capabilities in the second increment;
    - o spelling and grammar checking in the third increment;

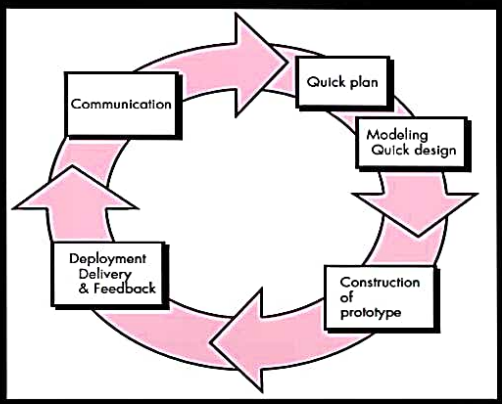
    and o advanced page layout capability in the fourth increment.
    - o It should be noted that the process flow for any increment can incorporate the prototyping paradigm.
- When an incremental model is used, the first increment is often a *core product*. That is, basic requirements are addressed but many supplementary features remain undelivered. The core product is used by the customer. As a result of use evaluation, a plan is developed for the next increment.
- The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality.
- This process is repeated following the delivery of each increment, until the complete product is produced.
- The incremental process model focuses on the delivery of an operational product with each increment.
- Early increments are stripped-down versions of the final product, but they do provide capability that serves the user and also provide a platform for evaluation by the user.
- Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project.
- Early increments can be implemented with fewer people. If the core product is well received, then additional staff (if required) can be added to implement the next increment.

Evolutionary Process Models

Evolutionary process models produce with each iteration produce an increasingly more complete version of the software with every iteration.

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

Prototyping.



- Prototyping is more commonly used as a technique that can be implemented within the context of anyone of the process model.

- The prototyping paradigm begins with communication. The software engineer and customer meet and define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory.
- Prototyping iteration is planned quickly and modeling occurs. The quick design leads to the construction of a prototype. The prototype is deployed and then evaluated by the customer/user.
- Iteration occurs as the prototype is tuned to satisfy the needs of the customer, while at the same time enabling the developer to better understand what needs to be done.

Example:

- If a customer defines a set of general objectives for software, but does not identify detailed input, processing, or output requirements, in such situation *prototyping* paradigm is best approach.
- If a developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system then he can go for this *prototyping* method.

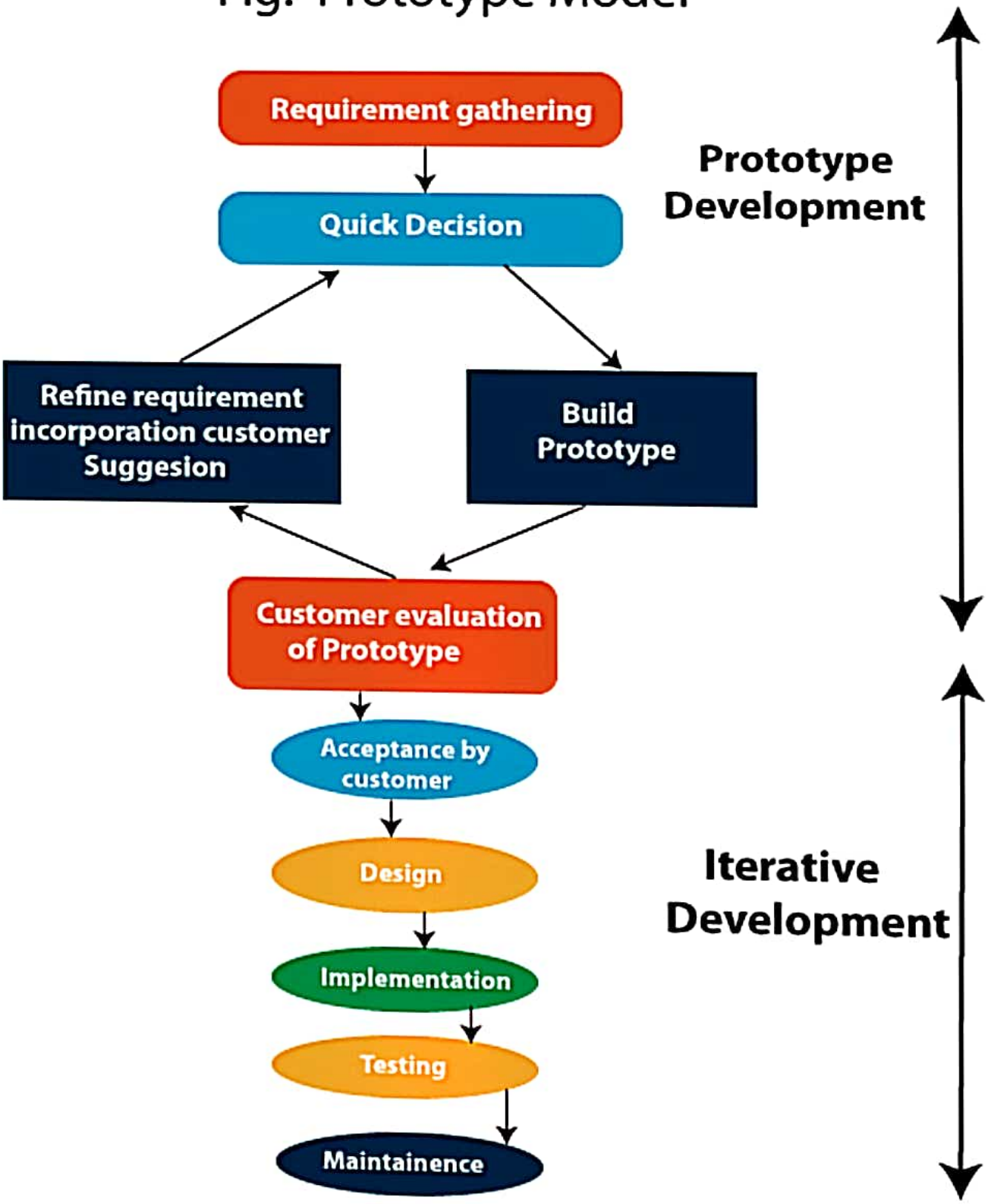
Advantages:

- The prototyping paradigm assists the software engineer and the customer to better understand what is to be built when requirements are fuzzy.
- The prototype serves as a mechanism for identifying software requirements. If a working prototype is built, the developer attempts to make use of existing program fragments or applies tools.
- Prototyping can be **problematic** for the following reasons:
- The customer sees what appears to be a working version of the software, unaware that the prototype is held together "with chewing gum and baling wire", unaware that in the rush to get it working we 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, the customer cries foul and demands that "a few fixes" be applied to make the prot[illegible] 32 / 47 [illegible]g product. Too often, software development relents.
- The developer often makes implementation compromises in order to get a prototype working quickly. An inappropriate operating system or programming language may be used simply because it is available and known; an inefficient algorithm may be implemented simply to demonstrate capability. After a time, the developer 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.

Prototype Model

The prototype model requires that before carrying out the development of actual software, a working prototype of the system should be built. A prototype is a toy implementation of the system. A prototype usually turns out to be a very crude version of the actual system, possible exhibiting limited functional capabilities, low reliability, and inefficient performance as compared to actual software. In many instances, the client only has a general view of what is expected from the software product. In such a scenario where there is an absence of detailed information regarding the input to the system, the processing needs, and the output requirement, the prototyping model may be employed.

Fig: Prototype Model



Steps of Prototype Model

1. Requirement Gathering and Analyst
2. Quick Decision
3. Build a Prototype
4. Assessment or User Evaluation
5. Prototype Refinement
6. Engineer Product

Advantage of Prototype Model

1. Reduce the risk of incorrect user requirement
2. Good where requirement are changing/uncommitted
3. Regular visible process aids management
4. Support early product marketing
5. Reduce Maintenance cost.
6. Errors can be detected much earlier as the system is made side by side.

1. An unstable/badly implemented prototype often becomes the final product.
2. Require extensive customer collaboration
    - Costs customer money
    - Needs committed customer
    - Difficult to finish if customer withdraw
    - May be too customer specific, no broad market
3. Difficult to know how long the project will last.
4. Easy to fall back into the code and fix without proper requirement analysis, design, customer evaluation, and feedback.
5. Prototyping tools are expensive.
6. Special tools & techniques are required to build a prototype.
7. It is a time-consuming process.