UNIT I

- UNIT-I:
- Introduction to Software Engineering:
- **A Generic view of process:** Software engineering, Process Framework, CMM, Process patterns, process assessment, Personal and Team process, Process Technology, Product and process.
- **Process models:** Perspective Models, The waterfall model, Incremental process models, Evolutionary process models, Specialized Process Models, The Unified process.
- **An Agile View of Process**: What is agility, Agile Process, and Agile Process Models

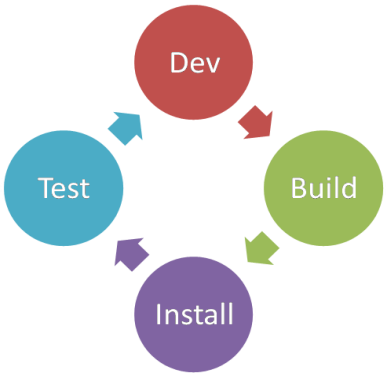
Introduction to Software Engineering

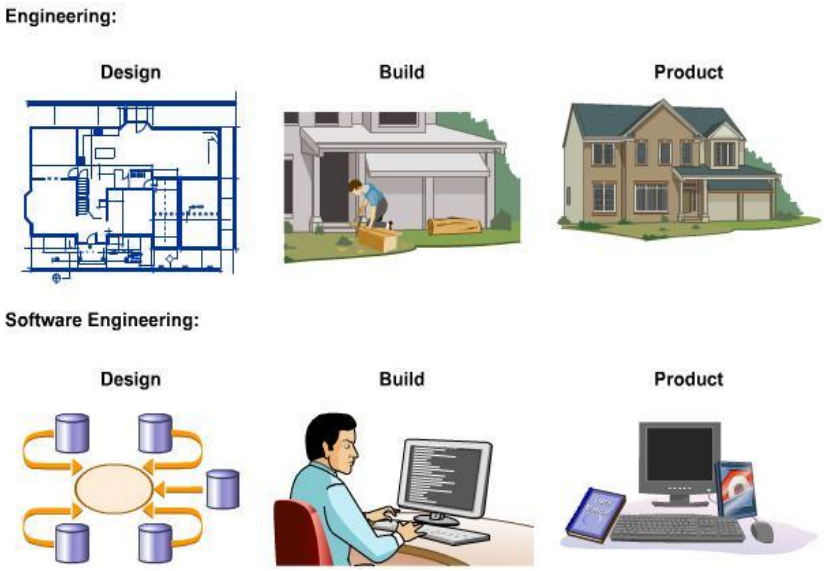
Software engineering is an engineering discipline that is concerned with all aspects of software development and production. We can alternatively view it as a systematic collection of past experience. The experience is arranged in the form of methodologies and guidelines. Software engineers should adopt a systematic and organised approach to their work and use appropriate tools and techniques depending on the problem to be solved, the development constraints and the resources available.

Definition of SOFTWARE ENGINEERING

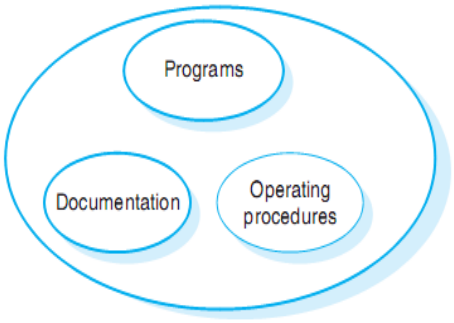
SE is defined as systematic, disciplined and quantifiable approach for the development, operation and maintenance of software by applying engineering principles.

- Software engineering can be defined as “The establishment and use of sound engineering principles in order to obtain economically software that is reliable and woks efficiently on real time machines.”





Software is defined as computer programs, procedures, rules and possibly associated documentation and data pertaining to the operation of a computer based systems. “Computer Software” is synonymous with “software product”.



Software = Program + Documentation + Operating Procedures

Another definition of Software is

- Instructions
  - Programs that when executed provide desired function
- Data structures
  -Enable the programs to adequately manipulate information
- Documents
  -Describe the operation and use of the programs

Characteristics of software

- Software is developed or engineered;
- it is not manufactured in the classical sense.
- Software does not wear out. However it deteriorates due to change.
- Software is custom built rather than assembling existing components.

Categories of Software

- Seven Broad Categories of software are challenges for software engineers
- System software
- Application software
- Engineering and scientific software
- Embedded software
- Product-line software
- Web-applications
- Artificial intelligence software

Legacy software are older programs that are developed decades ago. The quality of legacy software is poor because it has inextensible design, convoluted code, poor and nonexistent documentation, test cases and results that are not achieved.

As time passes legacy systems evolve due to following reasons:

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

Software Myths

Software Myths- beliefs about software and the process used to build it - can be traced to the earliest days of computing. Myths have a number of attributes that have made them insidious. For instance, myths appear to be reasonable statements of fact, they have an intuitive feel, and they are often promulgated by experienced practitioners who "know the score".

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. A software manager often grasps at belief in a software myth, If the Belief will lessen the pressure.

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 these entire question is no.

Myth : *If we get behind schedule, we can add more programmers and catch up (sometimes called the Mongolian horde concept)*

Reality : Software development is not a mechanistic process like manufacturing. In the words of 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

Myth : *If we 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

project internally, it will invariably struggle when it out sources software project.

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 led to false expectations and ultimately, dissatisfaction with the developers.

Myth : *A general statement of objectives is sufficient to begin writing programs we can fill in 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 are developed only through effective and continuous communication between customer and developer.

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

Reality : It's true that software requirement change, but the impact of change varies with the time at which it is introduced. When requirement changes are requested early, cost impact is relatively small. However, as time passes, cost impact grows rapidly – resources have been committed, a design framework has been established, and change can cause upheaval that requires additional resources and major design modification.

Practitioner Myths

Myth(1) → Once we write the program and get it to work, our job is done"

Reality → 60% to 80% of all effort expended on software occurs after it is delivered

Myth(2) → Until I get the program running, I have no way of assessing its quality

Reality → Formal technical reviews of requirements analysis documents, design documents, and source code (more effective than actual testing)

Myth(3) → The only deliverable work product for a successful project is the working program"

Reality → Software, documentation, test drivers, test results

Myth(4) → Software engineering will make us create voluminous and unnecessary documentation and will invariably slow us down

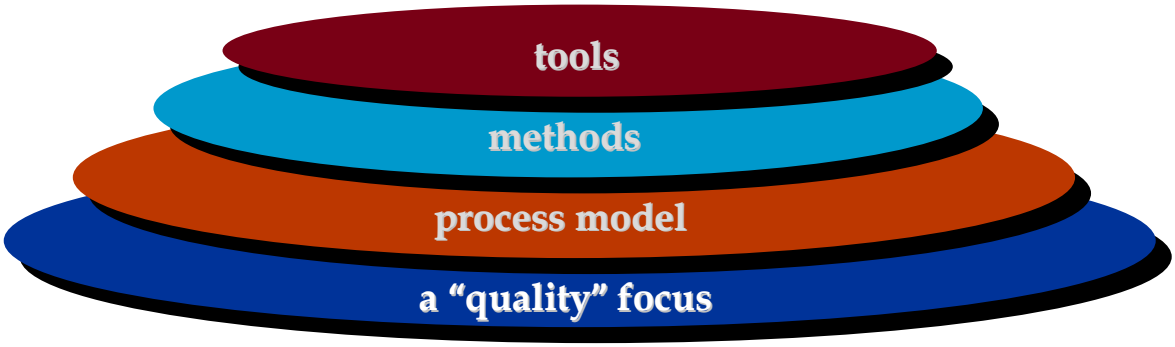
Reality →Creates quality, not documents; quality reduces rework and provides software on time and within the budget

A Generic view of process

Process: A set of activities, methods, practices, and transformations that people use to develop and maintain software and the associated products (e.g., project plans, design documents, code, test cases, and user manuals)

Software Engineering - A Layered Technology

Software engineering encompasses a process, the management of activities, technical methods, and use of tools to develop software products



- The foundation for software engineering is the *process* layer. It is the glue that holds the technology layers together and enables rational and 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, etc.) are produced, milestones are established, quality is ensured, and change is properly managed.
- Software engineering *methods* provide the technical "how to's" for building software. Methods encompass a broad array of tasks that include communication, req. analysis, design, coding, testing and support.
- Software engineering *tools* provide automated or semi-automated support for the process and the methods.

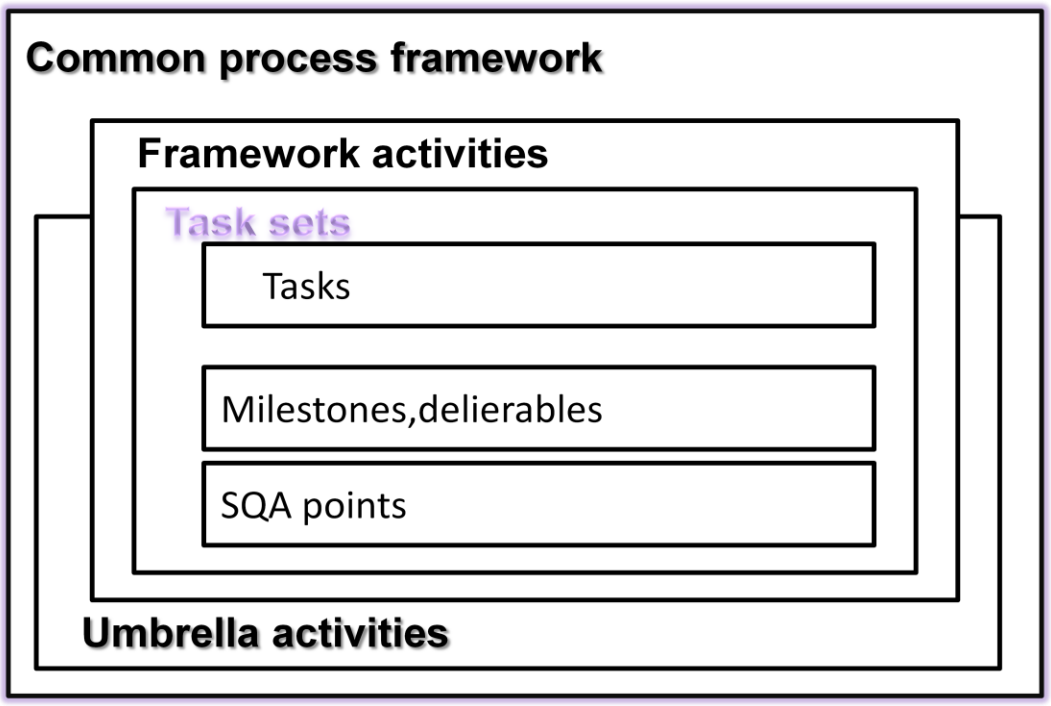
- When tools are integrated so that info. Created by one tool can be used by another, a system for the support of software development called computer-aided software engineering is established

A PROCESS FRAMEWORK

- Establishes the foundation for a complete software process
- Identifies a number of framework activities applicable to all software projects
- Also include a set of umbrella activities that are applicable across the entire software process.

- Used as a basis for the description of process models

- Generic process activities
  - Communication
  - Planning
  - Modeling
  - Construction
  - Deployment
  -

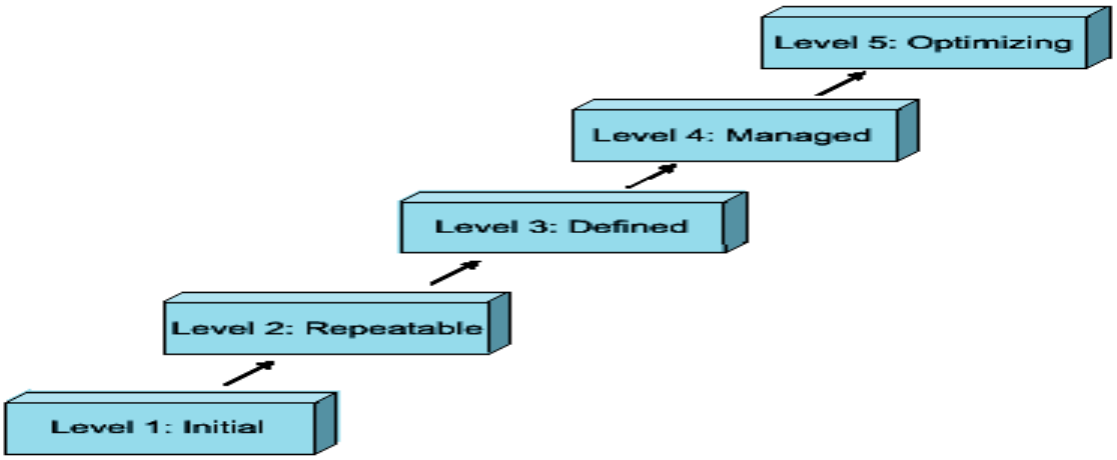


Communication activity
Planning activity
Modeling activity
- analysis action
- requirements gathering work task
- elaboration work task
- negotiation work task
- specification work task
- validation work task
- design action
- data design work task
- architectural design work task
- interface design work task
- component-level design work task
- Construction activity
- Deployment activity

- **Umbrella activities (examples)**

  - software project tracking and control
  - risk management
  - software quality assurance
  - formal technical reviews
  - measurement
  - s/w configuration management
  - reusability management
  - work product preparation and production (e.g., models, documents, logs)

CMM Levels.



Level 1: Initial.

- A software development organization at this level is characterized by ad hoc activities.
- Very few or no processes are defined and followed.
- Since software production processes are not defined, different engineers follow their own process and as a result development efforts become chaotic.
- The success of projects depends on individual efforts and heroics.
- Since formal project management practices are not followed, under time pressure short cuts are tried out leading to low quality.

Level 2: Repeatable

- At this level, the basic project management practices such as tracking cost and schedule are established.
- Size and cost estimation techniques like function point analysis, COCOMO, etc. are used.
- The necessary process discipline is in place to repeat earlier success on projects with similar applications. Opportunity to repeat a process exists only when a company produces a family of products

Level 3: Defined

- At this level the processes for both management and development activities are defined and documented.
- There is a common organization-wide understanding of activities, roles, and responsibilities.
- The processes though defined, the process and product qualities are not measured.
- ISO 9000 aims at achieving this level.

Level 4: Managed

- At this level, the focus is on software metrics.
- Two types of metrics are collected.
    - Product metrics measure the characteristics of the product being developed, such as its size, reliability, time complexity, understandability, etc.
    - Process metrics reflect the effectiveness of the process being used, such as average defect correction time, productivity, average number of defects found per hour inspection, average number of failures detected during testing per LOC, etc.
- Quantitative quality goals are set for the products. The software process and product quality are measured and quantitative quality requirements for the product are met.
- Various tools like Pareto charts, fishbone diagrams, etc. are used to measure the product and process quality.
- Thus, the results of process measurements are used to evaluate project performance rather than improve the process.

Level 5: Optimizing

- Process and product measurement data are analyzed for continuous process improvement.
- The process may be fine tuned to make the review more effective.
- The lessons learned from specific projects are incorporated in to the process.
- Continuous process improvement is achieved both by carefully analyzing the quantitative feedback from the process measurements and also from application of innovative ideas and technologies.
- These best practices are transferred throughout the organization.

Key process areas (KPA):

Each maturity level is characterized by several Key Process Areas (KPAs) except for SEI CMM level 1 that includes the areas an organization should focus to improve its software process to the next level.

CMM Level	Focus	Key Process Ares
1. Initial	Competent people	
2. Repeatable	Project management	Software project planning Software configuration management
3. Defined	Definition of processes	Process definition Training program Peer reviews
4. Managed	Product and process quality	Quantitative process metrics Software quality management
5. Optimizing	Continuous process improvement	Defect prevention Process change management Technology change management

Process Patterns

- Process patterns define a set of activities, actions, work tasks, work products and/or related behaviors
- A template is used to define a pattern
- Typical examples:
  - Customer communication (a process activity)
  - Analysis (an action)
  - Requirements gathering (a process task)
  - Reviewing a work product (a process task)
  - Design model (a work product)

Process Assessment

- The process should be assessed to ensure that it meets a set of basic process criteria that have been shown to be essential for a successful software engineering.

The generic process framework – Detailed Activities of each phase

- Communication
- Planning
- Modeling
- Construction
- Deployment

Communication Practices

Principles

- Listen
- Prepare before you communicate
- Facilitate the communication
- Face-to-face is best
- Take notes and document decisions
- Collaborate with the customer
- Stay focused
- Draw pictures when things are unclear
- Move on …
- Negotiation works best when both parties win.
- Initiation
    - The parties should be physically close to one another
    - Make sure communication is interactive
    - Create solid team “ecosystems”
    - Use the right team structure
- An abbreviated task set
    - Identify who it is you need to speak with
    - Define the best mechanism for communication
    - Establish overall goals and objectives and define the scope
    - Get more detailed
        - Have stakeholders define scenarios for usage
        - Extract major functions/features
    - Review the results with all stakeholders

Planning Practices

Principles

- Understand the project scope
- Involve the customer (and other stakeholders)
- Recognize that planning is iterative
- Estimate based on what you know
- Consider risk

- Be realistic
- Adjust granularity as you plan
- Define how quality will be achieved
- Define how you'll accommodate changes
- Track what you've planned
- Initiation
    - Ask Boehm's questions
        - Why is the system begin developed?
        - What will be done?
        - When will it be accomplished?
        - Who is responsible?
        - Where are they located (organizationally)?
        - How will the job be done technically and managerially?
        - How much of each resource is needed?
- An abbreviated task set
    - Re-assess project scope
    - Assess risks
    - Evaluate functions/features
    - Consider infrastructure functions/features
    - Create a coarse granularity plan
        - Number of software increments
        - Overall schedule
        - Delivery dates for increments
    - Create fine granularity plan for first increment
    - Track progress

Modeling Practices

- We create models to gain a better understanding of the actual entity to be built
- *Analysis models* represent the customer requirements by depicting the software in three different domains: the information domain, the functional domain, and the behavioral domain.

- *Design models* represent characteristics of the software that help practitioners to construct it effectively: the architecture, the user interface, and component-level detail.

Analysis Modeling Practices

- Analysis modeling principles
  - Represent the information domain
  - Represent software functions
  - Represent software behavior
  - Partition these representations
  - Move from essence toward implementation
- Elements of the analysis model
  - Data model
  - Flow model
  - Class model
  - Behavior model

Design Modeling Practices

- Principles
  - Design must be traceable to the analysis model
  - Always consider architecture
  - Focus on the design of data
  - Interfaces (both user and internal) must be designed
  - Components should exhibit functional independence
  - Components should be loosely coupled
  - Design representation should be easily understood
  - The design model should be developed iteratively
- Elements of the design model
  - Data design
  - Architectural design
  - Component design
  - Interface design

Construction Practices

- Preparation principles:
  - Understand of the problem you're trying to solve (see communication and modeling)
  - Understand basic design principles and concepts.

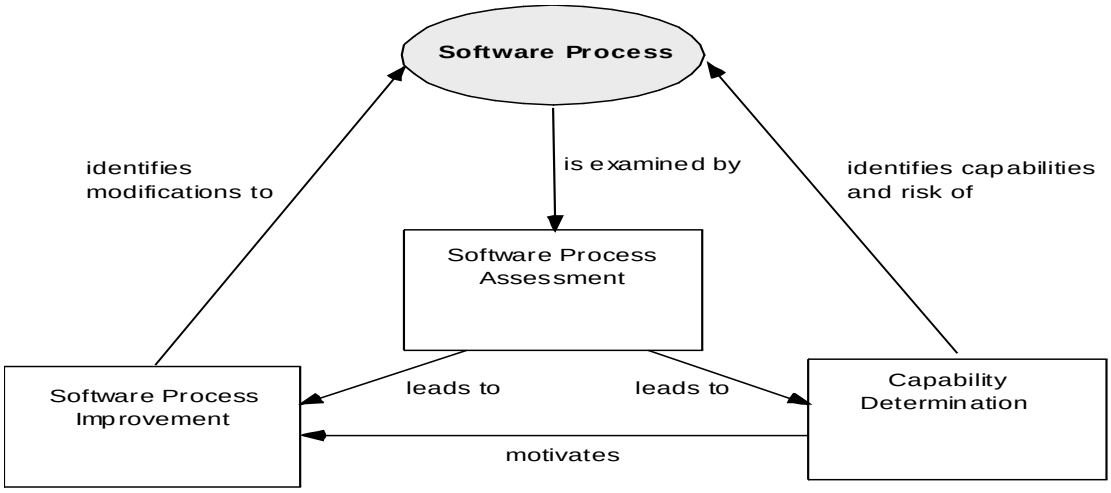
- Pick a programming language that meets the needs of the software to be built and the environment in which it will operate.
- Select a programming environment that provides tools that will make your work easier.
- Create a set of unit tests that will be applied once the component you code is completed**.**

- Coding principles: *After started writing code*
  - Constrain your algorithms by following structured programming practice.
  - Select data structures that will meet the needs of the design.
  - Understand the software architecture and create interfaces that are consistent with it.
  - Keep conditional logic as simple as possible.
  - Create nested loops in a way that makes them easily testable.
  - Select meaningful variable names and follow other local coding standards.
  - Write code that is self-documenting.
  - Create a visual layout (e.g., indentation and blank lines) that aids understanding.
- Validation Principles: *After completing first coding pass:*
  - Conduct a code walkthrough when appropriate.
  - Perform unit tests and correct errors you've uncovered.
  - Refactor the code
- Testing Principles
  - All tests should be traceable to requirements
  - Tests should be planned
  - The Pareto Principle applies to testing
  - Testing begins "in the small" and moves toward "in the large"
  - Exhaustive testing is not possible

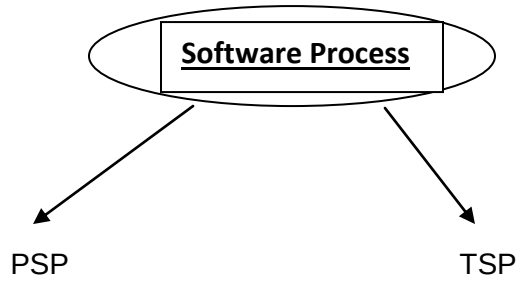
Deployment Practices

Principles

- Manage customer expectations for each increment
- A complete delivery package should be assembled and tested
- A support regime should be established
- Instructional materials must be provided to end-users
- Buggy software should be fixed first, delivered later



Software Process



Personal Software Process (PSP)

- Recommends five framework activities:
    - Planning
    - High-level design
    - High-level design review
    - Development
    - Postmortem
- stresses the need for each software engineer to identify errors early and as important, to understand the types of errors

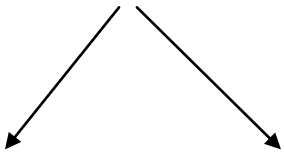
Team Software Process (TSP)

- Each project is "launched" using a "script" that defines the tasks to be accomplished
- Teams are self-directed
- Measurement is encouraged
- Measures are analyzed with the intent of improving the team process

PROCESS MODELS

1. PRESCRIPTIVE MODEL

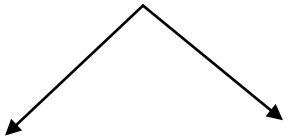
2. WATERFALL MODEL



CLASSICAL WATERFALL MODEL

ITERATIVE WATERFALL MODE

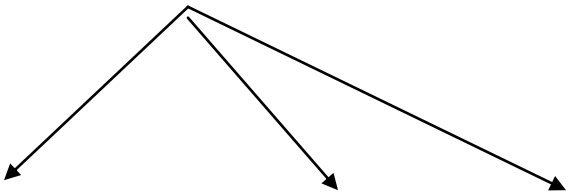
3. INCREMENTAL PROCESS MODELS



INCREMENTAL MODEL

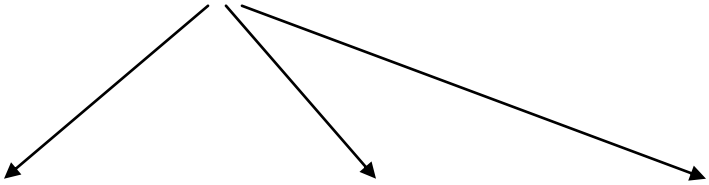
RAD (RAPID APPLICATION DEVELOPMENT MODEL)

4. EVOLUTIONARY PROCESS MODEL



PROTOTYPING MODEL SPIRAL MODEL CONCURRENT DEVELOPMENT MODEL

5. SPECIALISED PROCESS MODELS



CONCEPTUAL BASE DEVELOPMENT

FORMAL METHOD

ASPECT ORIENTED S/W DEVELOPMENT

6. THE UNIFIED PROCESS

Process Models

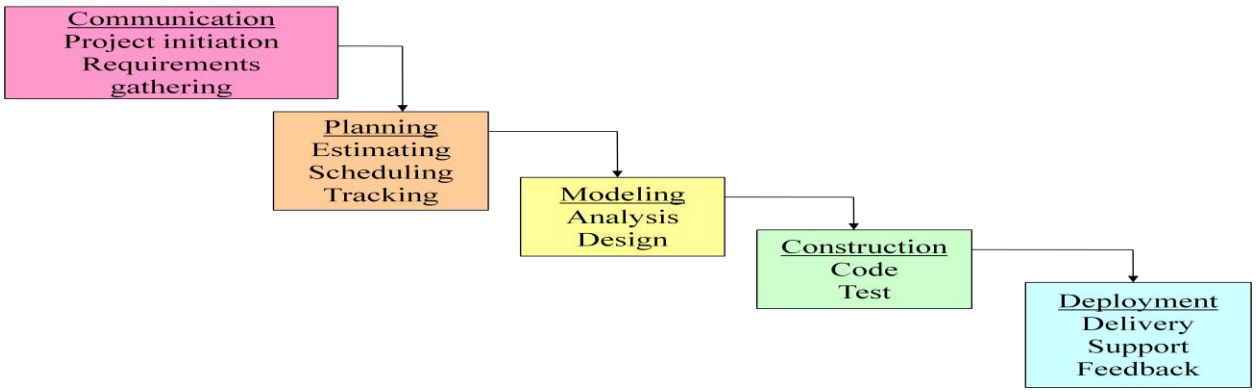
Life cycle model

A software life cycle model also called process model is a descriptive and diagrammatic representation of the software life cycle. A life cycle model represents all the activities required to make a software product transit through its life cycle phases. It also captures the order in which these activities are to be undertaken. In other words, a life cycle model maps the different activities performed on a software product from its inception to retirement.

Different life cycle models may map the basic development activities to phases in different ways. Thus, no matter which life cycle model is followed, the basic activities are included in all life cycle models though the activities may be carried out in different orders in different life cycle models. During any life cycle phase, more than one activity may also be carried out.

1. Classical Waterfall Model
2. Iterative Waterfall Model
3. Prototyping Model
4. Incremental Model
5. RAD Model
6. Spiral Model

1. Classical Waterfall Model



- Oldest software lifecycle model and best understood by upper management
- Used when requirements are well understood and risk is low
- Work flow is in a linear (i.e., sequential) fashion
- Used often with well-defined adaptations or enhancements to current software
- Begins with customer specification of Requirements and progresses through planning, modeling, construction and deployment

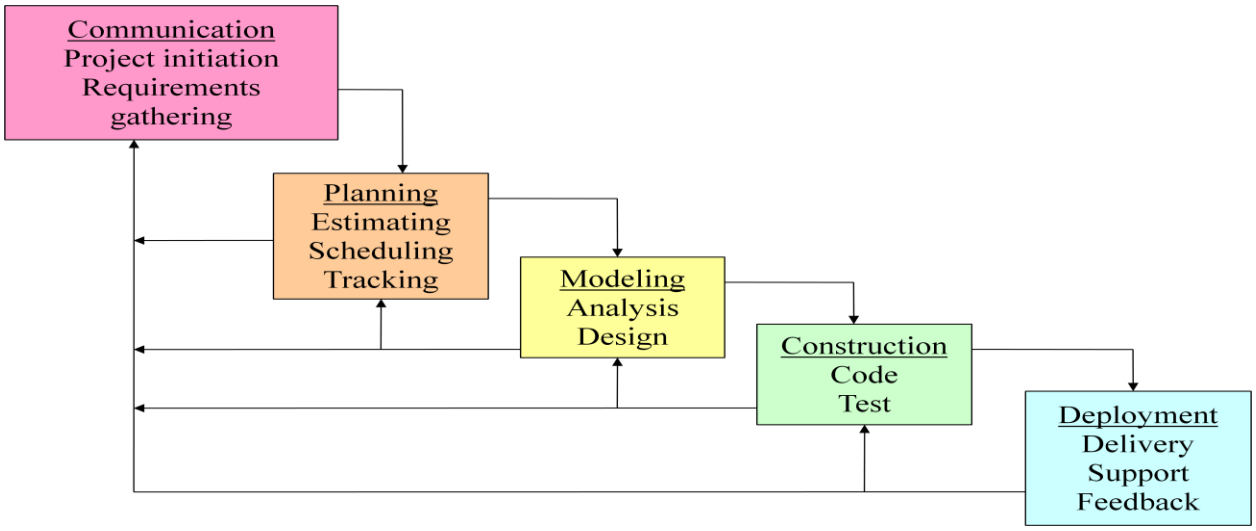
Advantages:

- It is very simple
- It divides the large task of building a software system into a seri es of clearly divided phases.
- Each phase is well documented

Problems

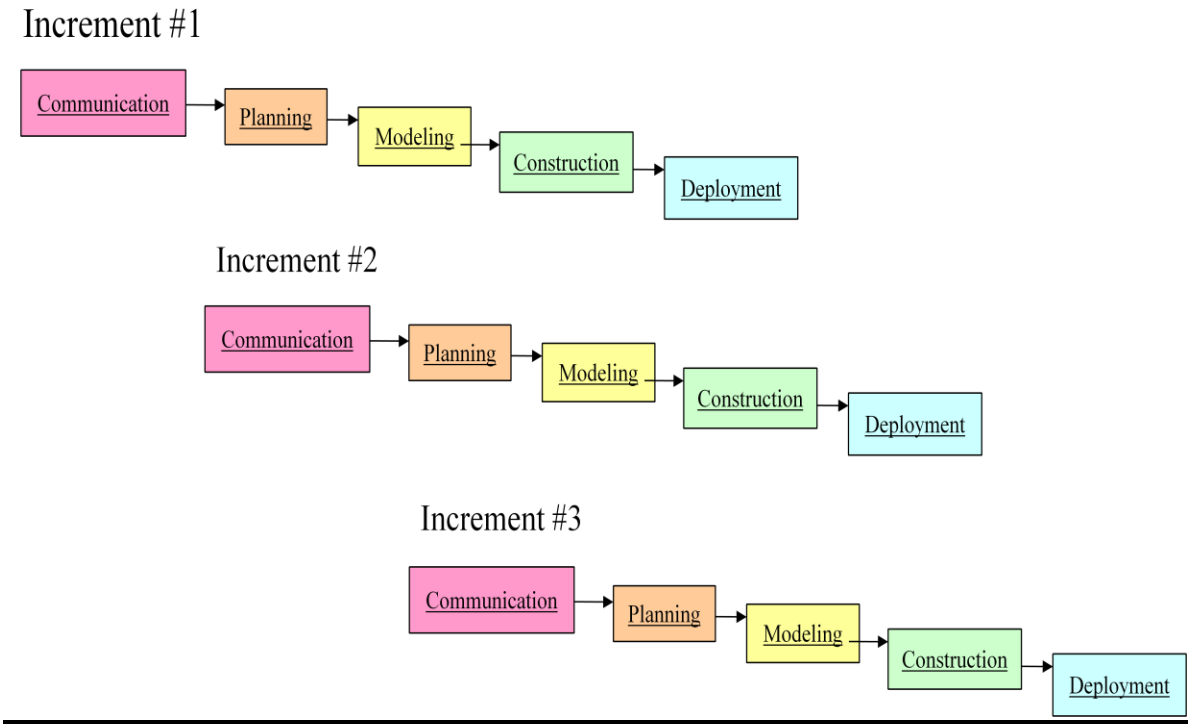
- Doesn't support iteration, so changes can cause confusion
- Difficult for customers to state all requirements explicitly and up front
- Requires customer patience because a working version of the program doesn't occur until the final phase
- Problems can be somewhat alleviated in the model through the addition of feedback loops

2. **Iterative Waterfall Model**



INCREMENTAL PROCESS MODELS

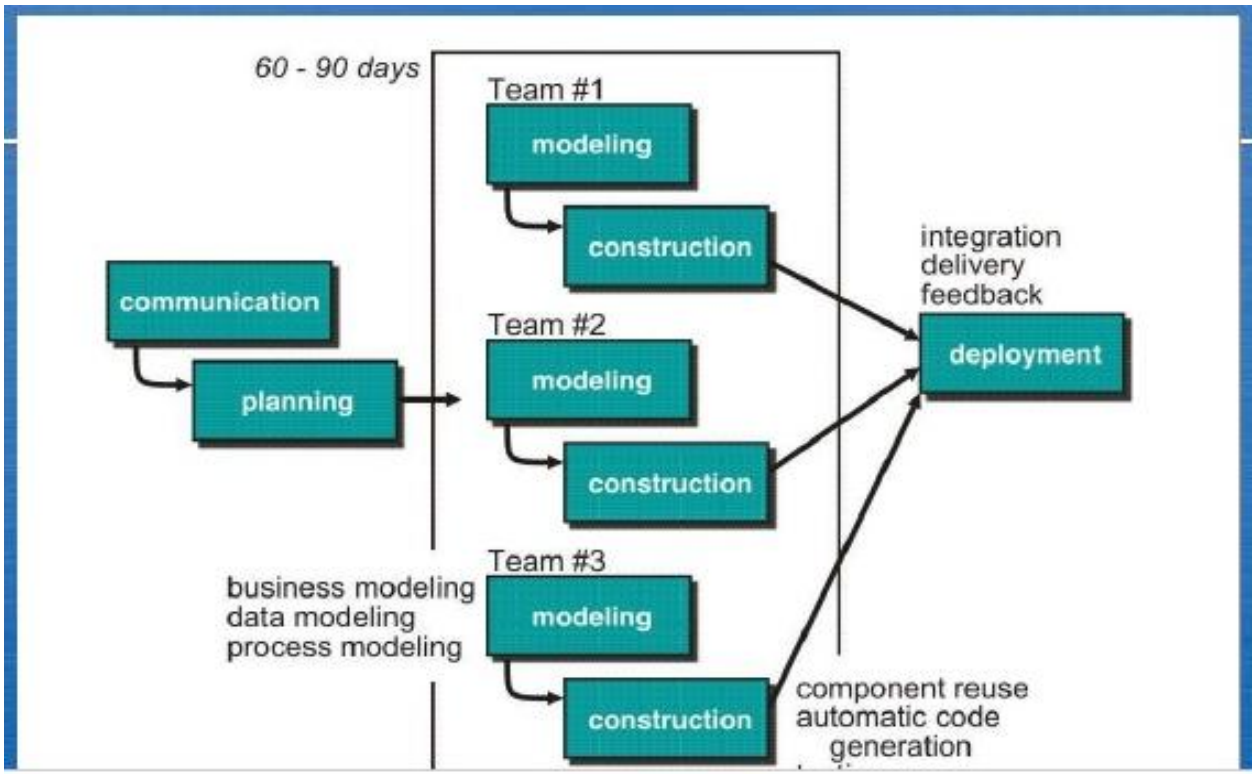
1. Incremental Model



- In this life cycle model, the software is first broken down into several modules which can be incrementally constructed and delivered.
- Used when requirements are well understood
- Multiple independent deliveries are identified
- Work flow is in a linear (i.e., sequential) fashion within an increment and is staggered between increments
- Iterative in nature; focuses on an operational product with each increment
- The development team first develops the core modules of the system.
- This initial product skeleton is refined into increasing levels of capability adding new functionalities in successive versions.
- Each evolutionary version may be developed using an iterative waterfall model of development.
- Provides a needed set of functionality sooner while delivering optional components later
- Useful also when staffing is too short for a full-scale development

2. Rapid Application Model (RAD)

- RAD is a **high speed** adaptation of linear sequential model. It is characterized by a very short development life cycle, in which the objective is to accelerate the development.
- The RAD model follows a component based approach.
- In this approach individual components developed by different people are assembled to develop a large software system.



The RAD model consist of the following phases

- Business Modeling:
  In this phase, define the flow of information within the organization, so that it covers all the functions. This helps in clearly understand the nature, type source and process of information.
- Data Modeling:
  In this phase, convert the component of the information flow into a set of data objects. Each object is referred as an *Entity.*
- Process Modeling:
  In this phase, the data objects defined in the previous phase are used to depict the flow of information . In addition adding , deleting, modifying and retrieving the data objects are included in process modeling.
- Application Designing:
  In this phase, the generation of the application and coding take place. Using fourth generation programming languages or 4 GL tools is the preferred choice for the software developers.
- Testing:
  In this phase, test the new program components.

The RAD has following advantages

- Due to emphasis on rapid development , it results in the delivery of fully functional project in short time period.
- It encourages the development of program component reusable.

The RAD has following disadvantages

- It requires dedication and commitment on the part of the developers as well as the client to meet the deadline. If either party is indifferent in needs of other, the project will run into serious problem.
- For large but scalable projects It is not suitable as RAD requires sufficient human resources to create the right number of RAD teams.
- RAD requires developers and customers who are committed to rapid fire activities
- Its application area is restricted to system that are modular and reusable in nature.
- It is not suitable for the applications that have a high degree of technical risk.
- For large but scalable projects, RAD requires sufficient human resources to create the right number of RAD teams.
- RAD requires developers and customers who are committed to rapid fire activities.
- Not all types of applications are appropriate for RAD.
- RAD is not appropriate when technical risks are high.

Evolutionary Process Models:

Prototype Models:

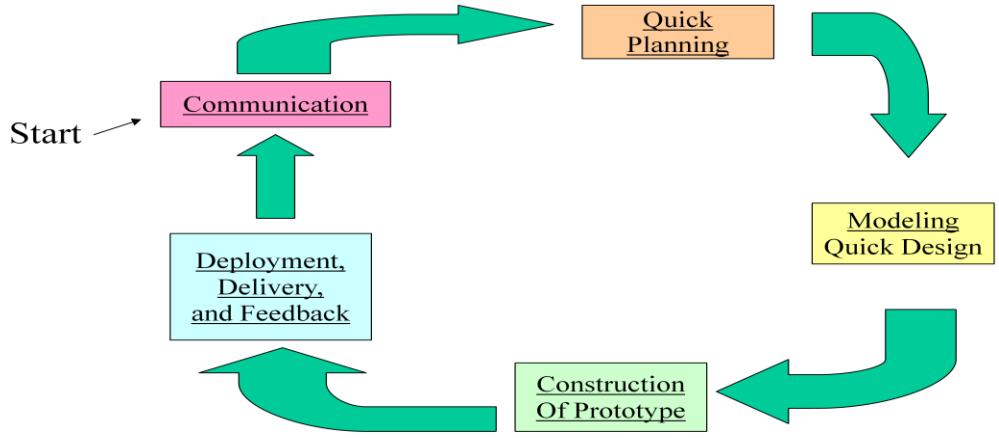
A prototype is a toy implementation of the system. A prototype usually exhibits limited functional capabilities, low reliability, and inefficient performance compared to the actual software. A prototype is usually built using several shortcuts. The shortcuts might involve using inefficient, inaccurate, or dummy functions. The shortcut implementation of a function, for example, may produce the desired results by using a table look-up instead of performing the actual computations.

Need for a prototype in software development

There are several uses of a prototype. An important purpose is to illustrate the input data formats, messages, reports, and the interactive dialogues to the customer. This is a valuable mechanism for gaining better understanding of the customer's needs:

- how the screens might look like
- how the user interface would behave
- how the system would produce outputs

Another reason for developing a prototype is that it is impossible to get the perfect product in the first attempt. Many researchers and engineers advocate that if you want to develop a good product you must plan to throw away the first version. The experience gained in developing the prototype can be used to develop the final product.



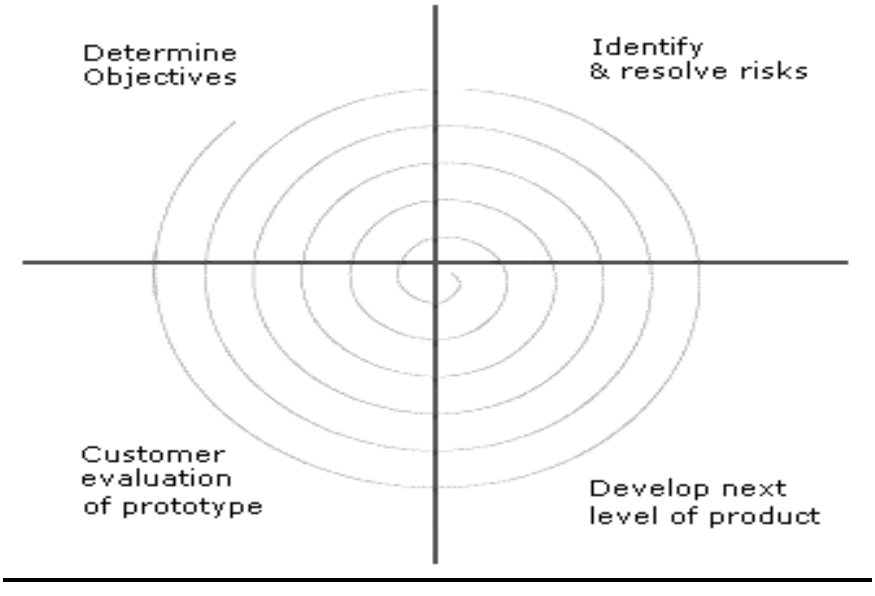
- Follows an evolutionary and iterative approach
- Used when requirements are not well understood
- Serves as a mechanism for identifying software requirements
- Focuses on those aspects of the software that are visible to the customer/user
- In this model, product development starts with an initial requirements gathering phase.
- A quick design is carried out and the prototype is built.
- The developed prototype is submitted to the customer for his evaluation.
- Based on the customer feedback, the requirements are refined and the prototype is suitably modified.
- This cycle of obtaining customer feedback and modifying the prototype continues till the customer approves the prototype.
- The actual system is developed using the iterative waterfall approach. However, in the prototyping model of development, the requirements analysis and specification phase becomes redundant as the working prototype approved by the customer becomes redundant as the working prototype approved by the customer becomes an animated requirements specification.

Disadvantages

- The customer sees a "working version" of the software, wants to stop all development and then buy the prototype after a "few fixes" are made
- Developers often make implementation compromises to get the software running quickly (e.g., language choice, user interface, operating system choice, inefficient algorithms)
- Lesson learned
  - Define the rules up front on the final disposition of the prototype before it is built
  - In most circumstances, plan to discard the prototype and engineer the actual production software with a goal toward quality
  -

Spiral Model

- Invented by Dr. Barry Boehm in 1988
- Follows an evolutionary approach
- Used when requirements are not well understood and risks are high
- Inner spirals focus on identifying software requirements and project risks; may also incorporate prototyping
- Outer spirals take on a classical waterfall approach after requirements have been defined, but permit iterative growth of the software
- Operates as a risk-driven model…a go/no-go decision occurs after each complete spiral in order to react to risk determinations
- Requires considerable expertise in risk assessment
- Serves as a realistic model for large-scale software development



First quadrant (Objective Setting)

- During the first quadrant, it is needed to identify the objectives of the phase.
- Examine the risks associated with these objectives.

Second Quadrant (Risk Assessment and Reduction)

- A detailed analysis is carried out for each identified project risk.
- Steps are taken to reduce the risks. For example, if there is a risk that the requirements are inappropriate, a prototype system may be developed.

Third Quadrant (Development and Validation)

- Develop and validate the next level of the product after resolving the identified risks.

Fourth Quadrant (Review and Planning)

- Review the results achieved so far with the customer and plan the next iteration around the spiral.
- Progressively more complete version of the software gets built with each iteration around the spiral.

Spiral Model Advantages

- Focuses attention on reuse options.
- It is a realistic approach to the development of large scale systems and software.
- Focuses attention on early error elimination.
- Puts quality objectives up front.
- Integrates development and maintenance.
- Provides a framework for hardware/software development.

Disadvantages:

- Contractual development often specifies process model
- and deliverables in advance.
- Requires risk assessment expertise.

Circumstances to use spiral model

The spiral model is called a meta model since it encompasses all other life cycle models. Risk handling is inherently built into this model. The spiral model is suitable for development of technically challenging software products that are prone to several kinds of risks. However, this model is much more complex than the other models – this is probably a factor deterring its use in ordinary projects.

Comparison of different life-cycle models

→The classical waterfall model can be considered as the basic model and all other life cycle models as embellishments of this model. However, the classical waterfall model can not be used in practical development projects, since this model supports no mechanism to handle the errors committed during any of the phases.

→This problem is overcome in the iterative waterfall model. The iterative waterfall model is probably the most widely used software development model evolved so far. This model is simple to understand and use. However, this model is suitable only for well-understood problems; it is not suitable for very large projects and for projects that are subject to many risks.

→The prototyping model is suitable for projects for which either the user requirements or the underlying technical aspects are not well understood. This model is especially popular for development of the user-interface part of the projects.

→The Incremental approach is suitable for large problems which can be decomposed into a set of modules for incremental development and delivery. This model is also widely used for object-oriented development projects. Of course, this model can only be used if the incremental delivery of the system is acceptable to the customer.

→The spiral model is called a meta model since it encompasses all other life cycle models. Risk handling is inherently built into this model. The spiral model is suitable for development of technically challenging software products that are prone to several kinds of risks. However, this model is much more complex than the other models – this is probably a factor deterring its use in ordinary projects.

The different software life cycle models can be compared from the viewpoint of the customer. Initially, customer confidence in the development team is usually high irrespective of the development model followed. During the lengthy development process, customer confidence normally drops off, as no working product is immediately visible.

Specialized Process Models

1. Component-based Development Model

- Consists of the following process steps
  - Available component-based products are researched and evaluated for the application domain in question
  - Component integration issues are considered
  - A software architecture is designed to accommodate the components
  - Components are integrated into the architecture
  - Comprehensive testing is conducted to ensure proper functionality

- Relies on a robust component library
- Capitalizes on software reuse, which leads to documented savings in project cost and time

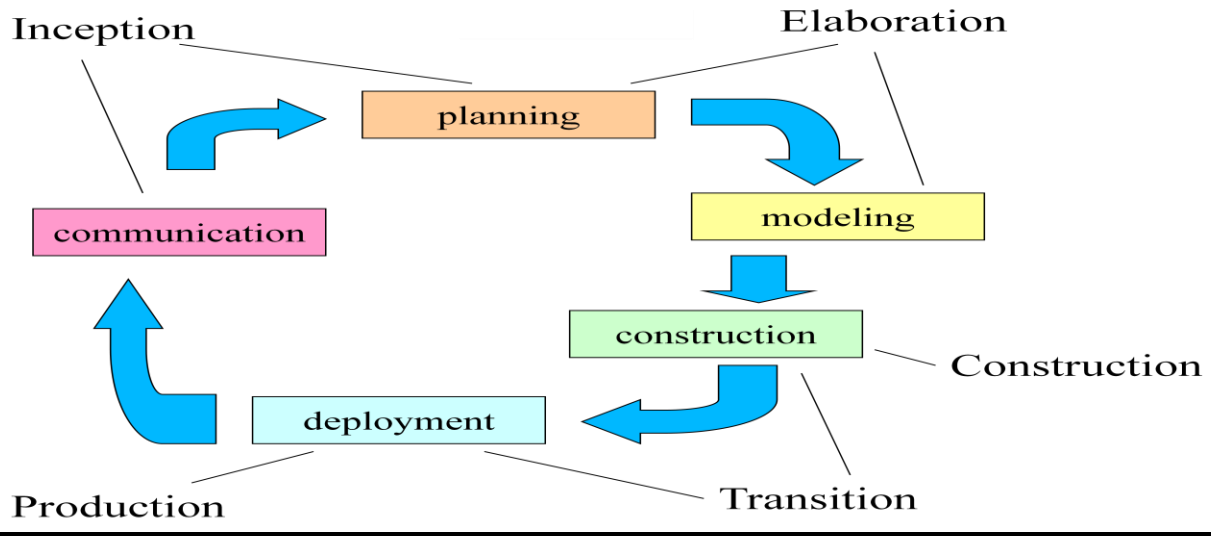
Formal Methods Model

- Encompasses a set of activities that leads to formal mathematical specification of computer software
- Enables a software engineer to specify, develop, and verify a computer-based system by applying a rigorous, mathematical notation
- Ambiguity, incompleteness, and inconsistency can be discovered and corrected more easily through mathematical analysis
- Offers the promise of defect-free software
- Used often when building safety-critical systems

Challenges of Formal Methods

- Development of formal methods is currently quite time-consuming and expensive
- Because few software developers have the necessary background to apply formal methods, extensive training is required
- It is difficult to use the models as a communication mechanism for technically unsophisticated customers

The Unified Process



- Birthed during the late 1980's and early 1990s when object-oriented languages were gaining wide-spread use
- Many object-oriented analysis and design methods were proposed; three top authors were Grady Booch, Ivar Jacobson, and James Rumbaugh
- They eventually worked together on a unified method, called the Unified Modeling Language (UML)
  - UML is a robust notation for the modeling and development of object-oriented systems

  - UML became an industry standard in 1997
  - However, UML does not provide the process framework, only the necessary technology for object-oriented development
- Booch, Jacobson, and Rumbaugh later developed the unified process, which is a framework for object-oriented software engineering using UML
  - Draws on the best features and characteristics of conventional software process models
  - Emphasizes the important role of software architecture
  - Consists of a process flow that is iterative and incremental, thereby providing an evolutionary feel
- Consists of five phases: inception, elaboration, construction, transition, and production

Inception Phase

- Encompasses both customer communication and planning activities of the generic process
- Business requirements for the software are identified
- A rough architecture for the system is proposed
- A plan is created for an incremental, iterative development
- Fundamental business requirements are described through preliminary use cases
  - A use case describes a sequence of actions that are performed by a user

Elaboration Phase

- Encompasses both the planning and modelling activities of the generic process
- Refines and expands the preliminary use cases
- Expands the architectural representation to include five views
  - Use-case model
  - Analysis model
  - Design model
  - Implementation model
  - Deployment model
- Often results in an executable architectural baseline that represents a first cut executable system
- The baseline demonstrates the viability of the architecture but does not provide all features and functions required to use the system

Construction Phase

- Encompasses the construction activity of the generic process
- Uses the architectural model from the elaboration phase as input
- Develops or acquires the software components that make each use-case operational

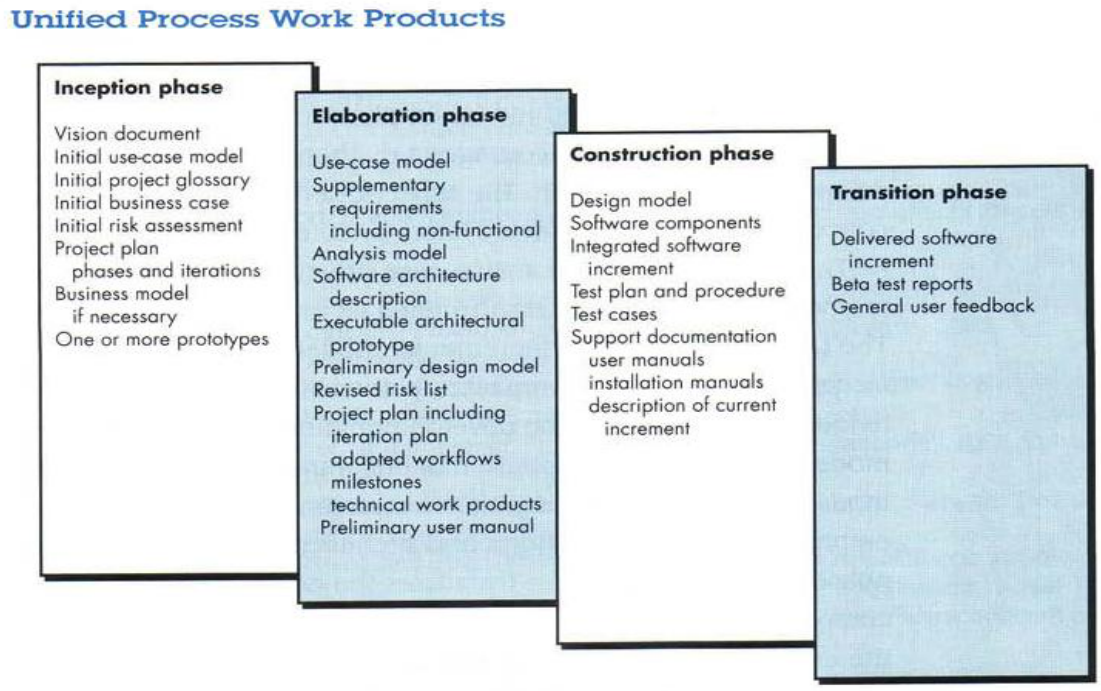
- Analysis and design models from the previous phase are completed to reflect the final version of the increment
- Use cases are used to derive a set of acceptance tests that are executed prior to the next phase

Transition Phase

- Encompasses the last part of the construction activity and the first part of the deployment activity of the generic process
- Software is given to end users for beta testing and user feedback reports on defects and necessary changes
- The software teams create necessary support documentation (user manuals, trouble-shooting guides, installation procedures)
- At the conclusion of this phase, the software increment becomes a usable software release

Production Phase

- Encompasses the last part of the deployment activity of the generic process
- On-going use of the software is monitored
- Support for the operating environment (infrastructure) is provided
- Defect reports and requests for changes are submitted and evaluated



Agile Processing

What is Agility

Agility has become today's buzzword when describing a modern software process. Everyone is agile. An agile team is a nimble team able to appropriately respond to changes. Change is what software development is very much about. Changes in the software being built, changes to the team members, changes because of new technology, changes of all kinds that may have an impact on the product they build or the project that creates the product. Support for changes should be built-in everything we do in software, something we embrace because it is the heart and soul of software. An agile team recognizes that software is developed by individuals working in teams and that the skills of these people, their ability to collaborate is at the core for the success of the project.

Agility

- Effective response to change
- Effective communication among all stakeholders
- Drawing the customer onto the team; eliminate the "us and them" attitude
- Organizing a team so that it is in control of the work performed
- Rapid, incremental delivery of software

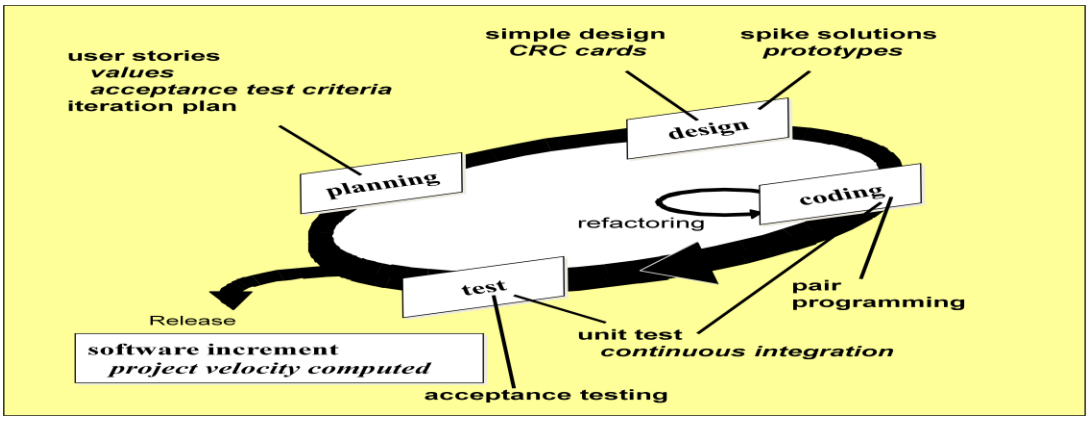
12 Principles to achieve agility – by the Agile Alliance

1. Highest priority -> satisfy the customer
2. Welcome changing requirements
3. Deliver working software frequently
4. Business people and developers must work together
5. Build projects around motivated individuals
6. Emphasize face-to-face conversation
7. Working software is the primary measure of progress
8. Agile processes promote sustainable development
9. Continuous attention to technical excellence and good design enhances agility
10. Simplicity – the art of maximizing the amount of work not done – is essential
11. The best designs emerge from self-organizing teams
12. The team tunes and adjusts its behavior to become more effective

Agile Process Models

1. Extreme Programming (XP)
2. Adaptive Software Development (ASD)
3. Dynamic Systems Development Method (DSDM)
4. Scrum
5. Crystal
6. Feature Driven Development (FDD)
7. Agile Modeling (AM)

1. Extreme Programming (XP)



- The most widely used agile process, originally proposed by Kent Beck [BEC99]
- XP uses an object-oriented approach as its preferred development paradigm
- Defines four (4) framework activities
  - Planning
  - Design
  - Coding
  - Testing

XP Planning

- Begins with the creation of “user stories” and then placed on an index card.
- The customer assigns a *value* to the story based on the overall business value of the function.
- Agile team assesses each story and assigns a cost “measured in development weeks.”
- Stories are grouped to form a deliverable increment
- A commitment is made on delivery date

Once a commitment is made on delivery date, the XP team orders the stories that will be developed in one of three ways:

1. All stories will be implemented immediately within a few weeks.
2. The stories with the highest value will be moved up in the schedule and implemented first.
3. The riskiest stories will be moved up in the schedule and implemented first.

– After the first increment (project release), "project velocity" is used to help define subsequent delivery dates for other increments.

– *Project velocity* is the number of customer stories implemented during the first release

XP – Design

- Follows the KIS (keep it simple) principle
- Encourage the use of CRC (class -responsibility - collaborator) cards
- For difficult design problems, suggests the creation of "*spike solutions*"—a design prototype that is implemented and evaluated
- Encourages "*refactoring*"—an iterative refinement of the internal program design that controls the code modifications by suggesting small design changes that may improve the design.
- Design occurs both before and after coding commences

XP – Coding

- Recommends the construction of a series of unit tests for each of the stories before coding commences
- Encourages "*pair programming*"
  - Mechanism for real-time problem solving and real-time quality assurance
  - Keeps the developers focused on the problem at hand
- Needs continuous integration with other portions (stories) of the s/w, which provides a "smoke testing" environment

XP – Testing

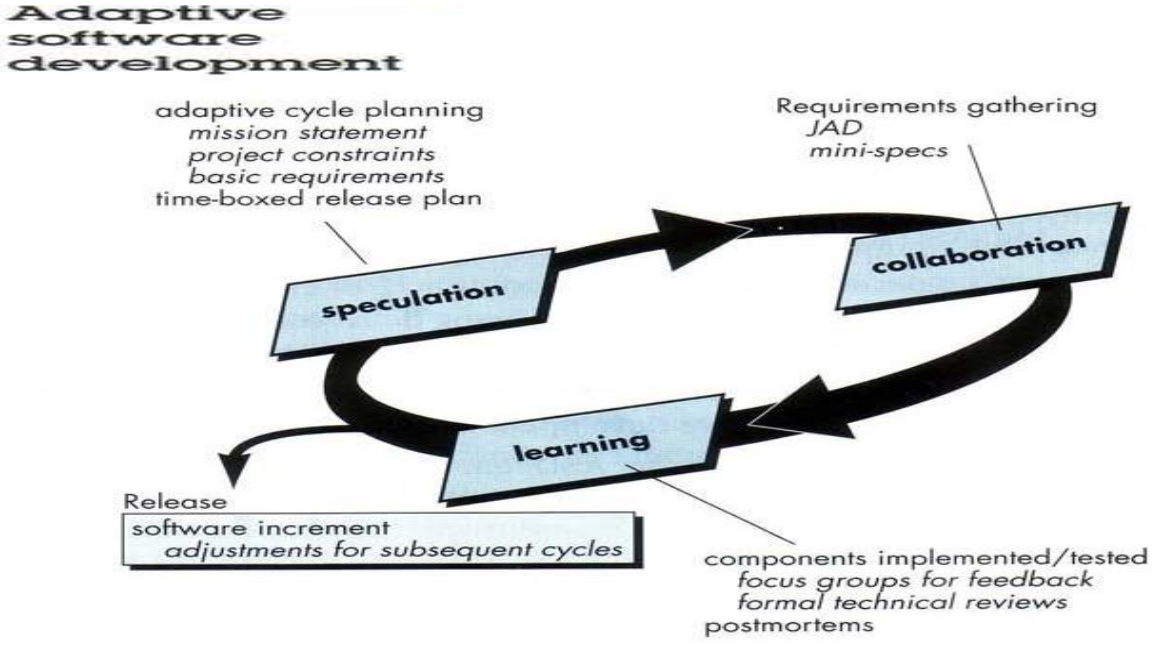
- Unit tests should be implemented using a framework to make testing automated. This encourages a regression testing strategy.
- Integration and validation testing can occur on a daily basis
- *Acceptance tests*, also called *customer tests*, are specified by the customer and executed to assess customer visible functionality
- Acceptance tests are derived from user stories

Advantages

- Customer focus increase the chance that the software produced will actually meet the needs of the users
- The focus on small, incremental release decreases the risk on your project:
  - by showing that your approach works and
  - by putting functionality in the hands of your users, enabling them to provide timely feedback regarding your work.
- Continuous testing and integration helps to increase the quality of your work
- XP is attractive to programmers who normally are unwilling to adopt a software process, enabling your organization to manage its software efforts better.

Disadvantages

- XP is geared toward a single project, developed and maintained by a single team.
- XP is particularly vulnerable to "bad apple" developers who:
  - don't work well with others
  - who think they know it all, and/or
  - who are not willing to share their "superior" code
- XP will not work in an environment where a customer or manager insists on a complete specification or design before they begin programming.
- XP will not work in an environment where programmers are separated geographically.
- XP has not been proven to work with systems that have scalability issues (new applications must integrate into existing systems).



- Originally proposed by Jim Highsmith
- ASD — distinguishing features
  - **Mission-driven** planning
  - **Component-based focus**
  - Uses **"time-boxing"**
  - Explicit consideration of **risks**
  - Emphasizes **collaboration** for requirements gathering
  - Emphasizes "**learning**" throughout the process

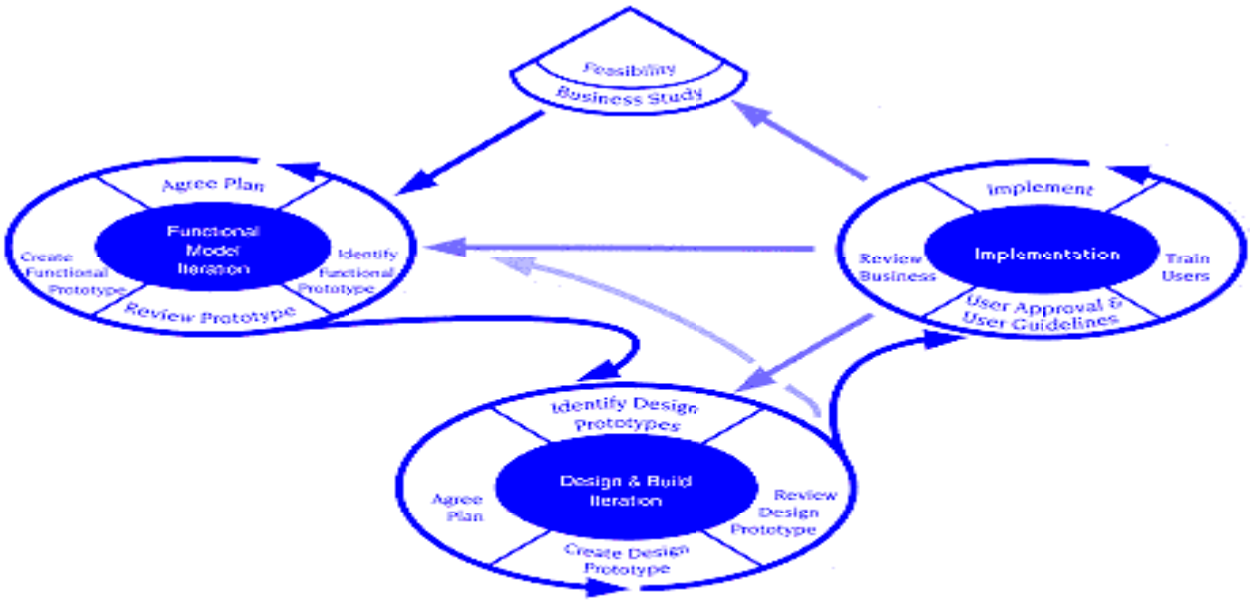
- **Speculation**: An adaptive cycle-planning is conducted where it uses the customer's mission statement, project constraints (delivery dates, user description) and basic requirements.
- **Collaboration**: People working together must trust one another to:
  1. criticize without animosity
  2. assist without resentment
  3. work as hard or harder as they do
  4. have the skill set to contribute to the work at hand
  5. communicate problems or concerns in a way that leads to effective action
- **Learning**
- Learning will help them to improve their level of real understanding.

Dynamic Systems Development Method

- *Dynamic System Development Method* is an agile S/W development approach that provides a framework for building and maintaining systems which meet tight time constraints through the use of incremental prototyping in a controlled project environment.
- Promoted by the DSDM Consortium
- DSDM—distinguishing features
- Nine guiding principles
  - Active user involvement is imperative.
  - DSDM teams must be empowered to make decisions.
  - The focus is on frequent delivery of products.
  - Fitness for business purpose is the essential criterion for acceptance of deliverables.
  - Iterative and incremental development is necessary to converge on an accurate business solution.
  - All changes during development are reversible.
  - Requirements are baselined at a high level
  - Testing is integrated throughout the life-cycle.

It defines three different iterative cycles preceded
by two additional life cycles activities:

- *Feasibility Study*: Business requirements and constraints.
- *Business Study:* Establishes req. that will allow the application to provide business value.
- *Functional Model Iteration*: Produce iterative prototypes.
- *Design and build iteration*: Revisit prototyping.
- *Implementation*: Include the latest prototype into the operational environment.



Scrum principles

- Small working teams are organized to "maximize communication, minimize overhead, and maximize sharing of tacit, informal knowledge."
- The process must be adaptable to both technical and business changes "to ensure the best possible product is produced."
- The process yields frequent software increments "that can be inspected, adjusted, tested, documented, and built on."
- Development work and the people who perform it are partitioned "into clean, low coupling partitions, or packets."
- Constant testing and documentation is performed as the product is built.

Development work is partitioned into "packets"

- Testing and documentation are on-going as the product is constructed
- Work occurs in "sprints ➔ framework activities" and is derived from a "backlog ➔ requirements that provide business values" of existing requirements
- Meetings are very short and sometimes conducted without chairs
- "Demos" are delivered to the customer with the time-box allocated

Development Activities of SCRUM

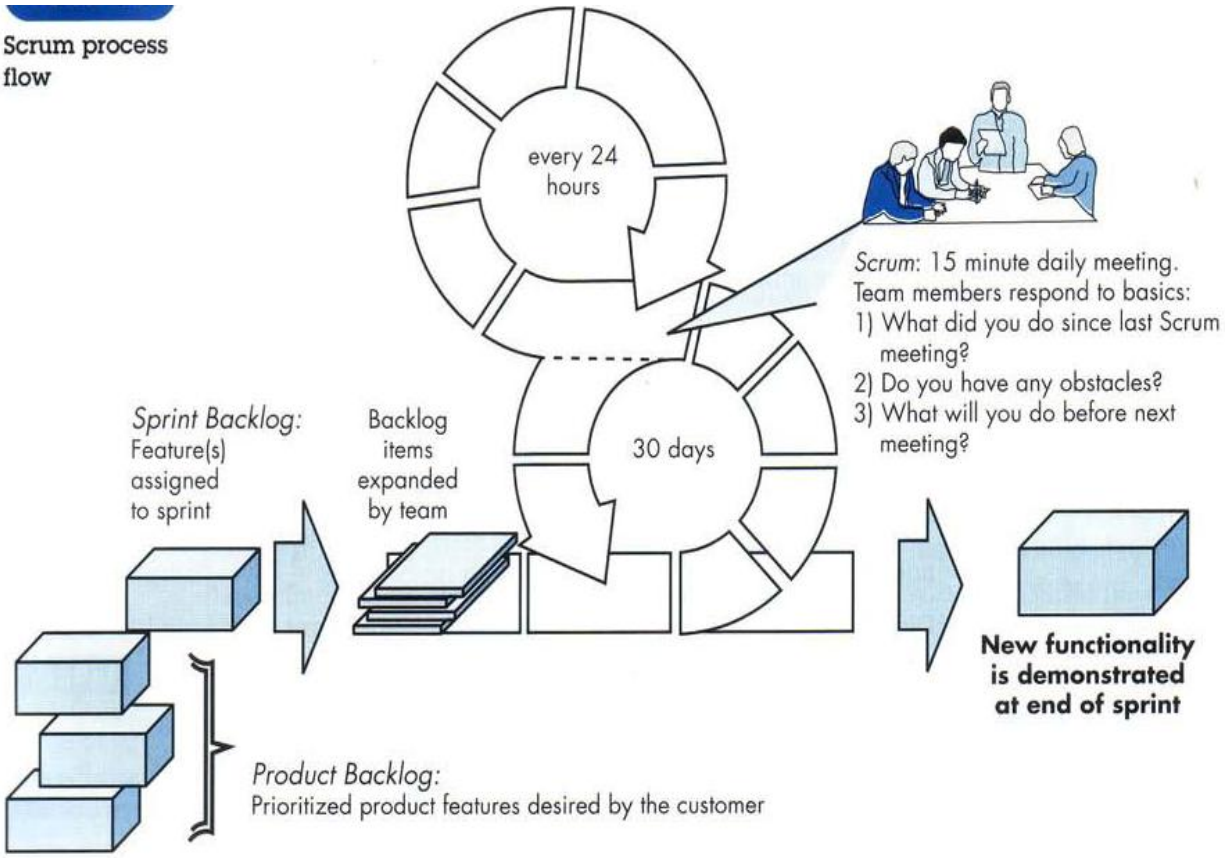
Backlog—a prioritized list of project requirements or features that provide business value for the customer. Items can be added to the backlog at any time (this is how changes are introduced). The product manager assesses the backlog and updates priorities as required.

Sprints—consist of work units that are required to achieve a requirement defined in the backlog that must be fit into a predefined time-box (typically 30 days). During the sprint, the backlog items that the sprint work units address are frozen (i.e., changes are not introduced during the sprint). Hence, the sprint allows team members to work in a short-term, but stable environment.

Scrum meetings—are short (typically 15 minutes) meetings held daily by the Scrum team. Three key questions are asked and answered by all team members

- What did you do since the last team meeting?
- What obstacles are you encountering?
- What do you plan to accomplish by the next team meeting?

Demos—deliver the software increment to the customer so that functionality that has been implemented can be demonstrated and evaluated by the customer. It is important to note that the demo may not contain all planned functionality, but rather those functions that can be delivered within the time-box that was established.

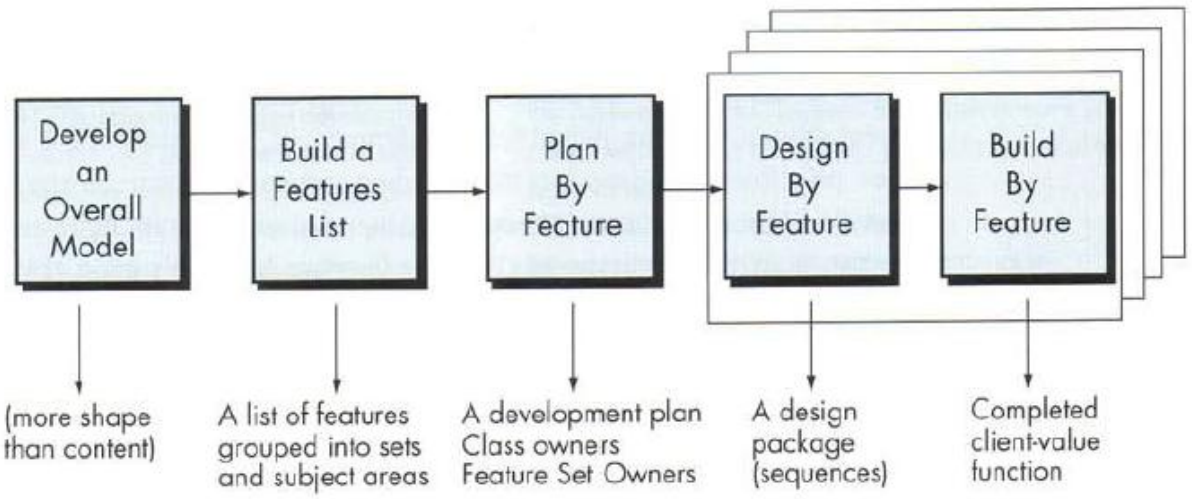


Crystal

- Proposed by Cockburn and Highsmith
- Crystal—distinguishing features
  - Actually a family of process models that allow "maneuverability" based on problem characteristics
  - Face-to-face communication is emphasized
  - Suggests the use of "reflection workshops" to review the work habits of the team

8. Feature Driven Development (FDD)

In the context of FDD, a *feature* "is a client-valued function that can be implemented in two weeks or less" The emphasis on the definition of features provides the following benefits:


Develop an Overall Model
Build a Features List
Plan By Feature
Design By Feature
Build By Feature
(more shape than content)
A list of features grouped into sets and subject areas
A development plan
Class owners
Feature Set Owners
A design package (sequences)
Completed client-value function