(1) Software & Software Engineering

Software:

- Computer software is the product that software professionals build and then support over the long term.
- It encompasses programs that execute within a computer of any size and architecture, content that is presented as the computer programs execute, and descriptive information in both hard copy and virtual forms that encompass virtually any electronic media.

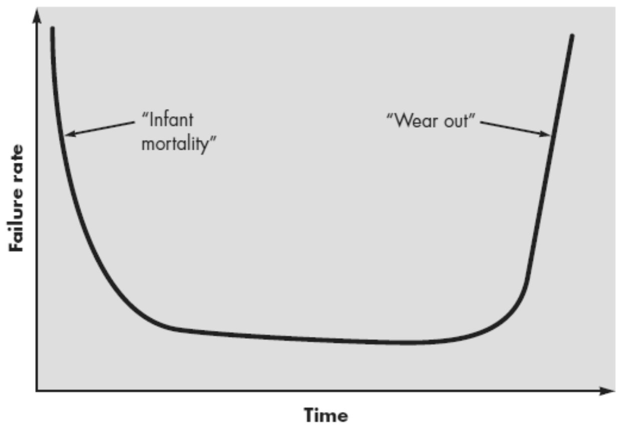
Characteristics of software :

[1] Software is developed or engineered; it is not manufactured in the classical sense:

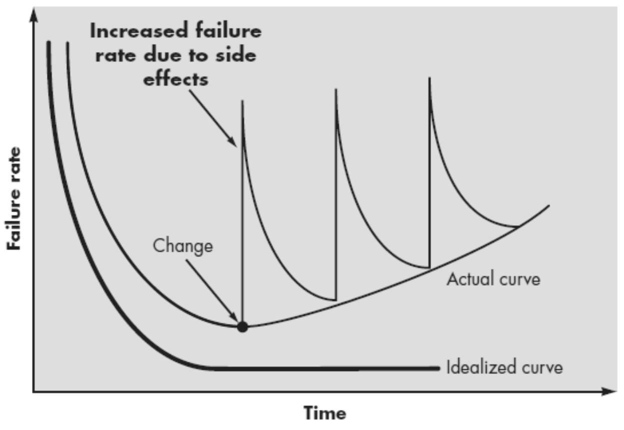
- Although some similarities exist between software development and hardware manufacturing, but few activities are fundamentally different.
- In both activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems than software.

[2] Software doesn't "wear out."

- Hardware components suffer from the growing effects of dust, vibration, abuse, temperature extremes, and many other environmental maladies. Stated simply, the hardware begins to wear out.
- Software is not susceptible to the environmental maladies that cause hardware to wear out. In theory, therefore, the failure rate curve for software should take the form of the "idealized curve".
- When a hardware component wears out, it is replaced by a spare part.
- There are no software spare parts.
- Every software failure indicates an error in design or in the process through which design was translated into machine executable code.
- Therefore, the software maintenance tasks that accommodate requests for change involve considerably more complexity than hardware maintenance.
- However, the implication is clear—software doesn't wear out. But it does deteriorate.



Hardware Failure curve



Software Failure curve

[3] Although the industry is moving toward component-based construction, most software continues to be custom built.

- A software component should be designed and implemented so that it can be reused in many different programs.

- Modern reusable components encapsulate both data and the processing that is applied to the data, enabling the software engineer to create new applications from reusable parts.
- In the hardware world, component reuse is a natural part of the engineering process

Difference between program and software

Program	Software
1) Small in size.	1) Large in size.
2) Authors himself is user-soul.	2) Large number.
3) Single developer.	3) Team developer.
4) Adopt development.	4) Systematic development.
5) Lack proper interface.	5) Well define interface.
6) Large proper documentation.	6) Well documented

Software Myths:

Software myths are beliefs about software and the process used to build it.

(1) Management Myths:

Myth 1:

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 2:

If we get behind schedule, we can add more programmers and catch up.

Reality:

- Software development is not a mechanistic process like manufacturing.
- 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 3:

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.

(2) Customer Myths:

Myth 1:

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 are developed only through effective and continuous communication between customer and developer.

Myth 2:

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 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 cause upheaval that requires additional resources and major design modification.

(3) Practitioner's Myths:

Myth 1:

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

Reality:

- 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 2:

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 3:

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 4:

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.

Software Engineering:

- Software Engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.

A Layered Technology:

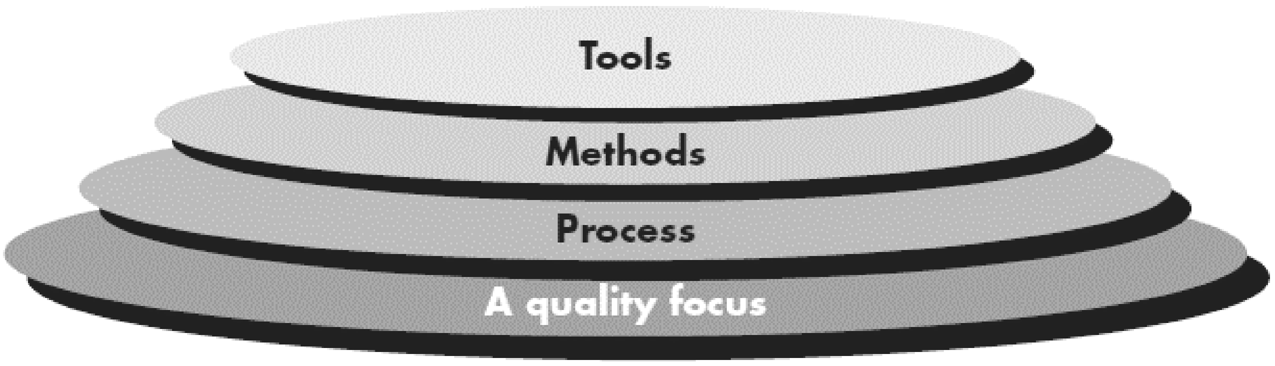


Figure: Software Engineering Layers

A quality Focus:

- Every organization is rest on its **commitment to quality.**
- Total **quality management, Six Sigma, or similar continuous improvement culture** and it is this culture ultimately leads to development of increasingly more effective approaches to software engineering.
- The foundation that supports software engineering is a quality focus.

Process:

- The software engineering process 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 are produced, milestones are established, quality is ensured, and change is properly managed.

Methods:

- Software engineering methods provide the technical aspects for building software.
- Methods encompass a broad array of tasks that include communication, requirements analysis, design modeling, program construction, testing, and support.
- Software engineering methods rely on the set of modeling activities and other descriptive techniques.

Tools:

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

(2) Process & Generic Process Model

- A software process is 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.

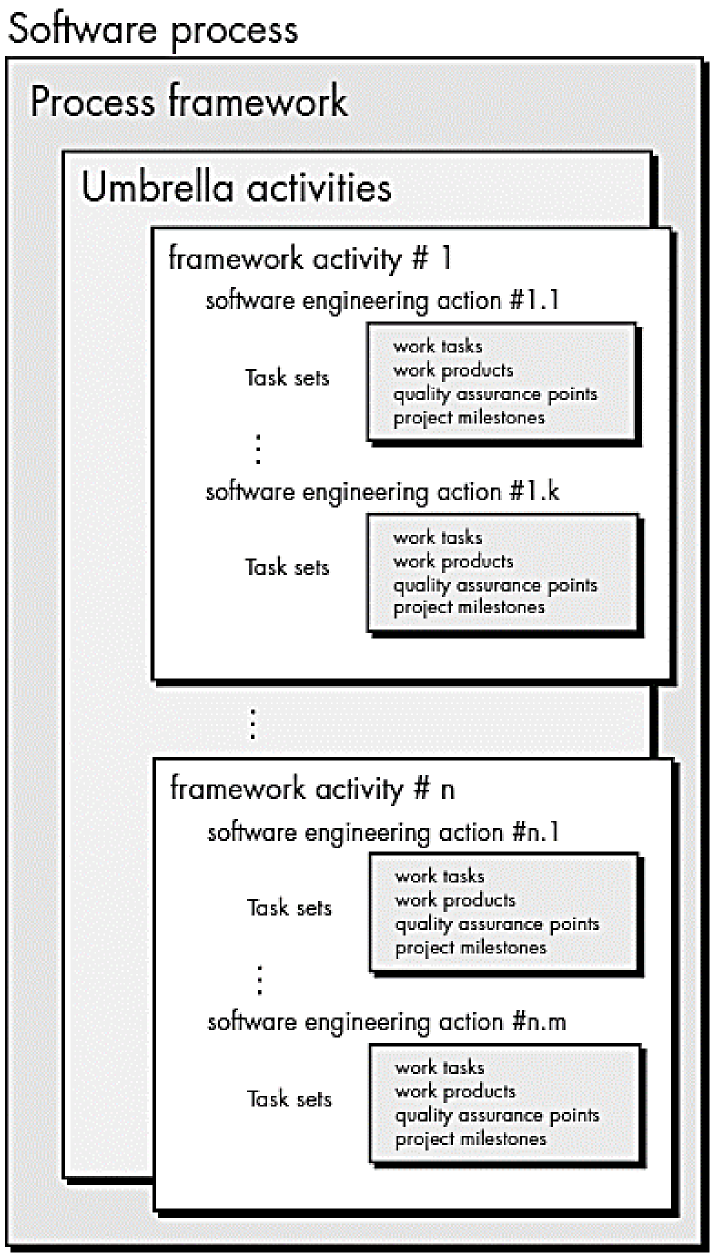
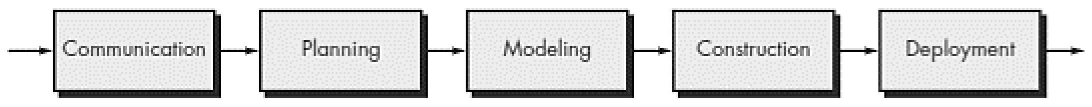


Figure: Generic Process Model

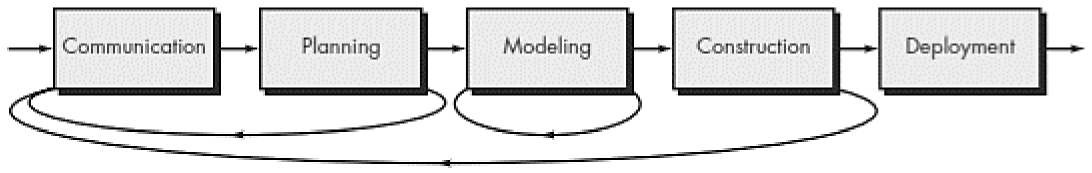
- Each framework activity is populated by a set of software engineering actions.
- Each software engineering action is defined by a task set that identifies the work tasks that are to be completed, the work products that will be produced, the quality assurance points that will be required, and the milestones that will be used to indicate progress.
- A generic process framework for software engineering defines **five framework activities—communication, planning, modeling, construction, and deployment**.
- In addition, it defines a set of umbrella activities.

***Process Flow*:**

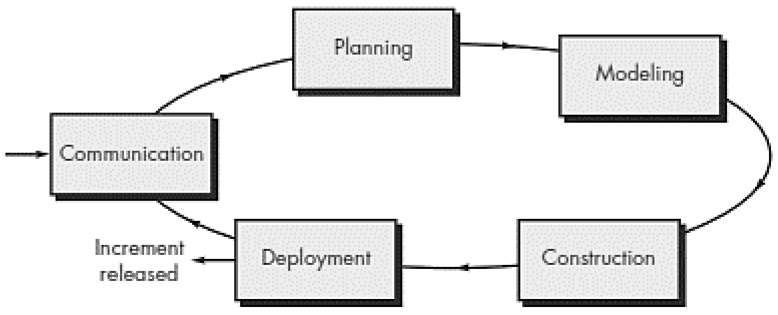
- **Process flow**—describes how the framework activities and the actions and tasks that occur within each framework activity are organized with respect to sequence and time.



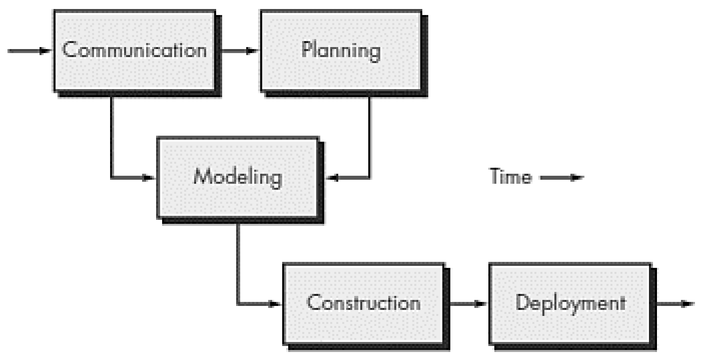
(a) Linear process flow



(b) Iterative process flow



(c) Evolutionary process flow



(d) Parallel process flow

- A linear process flow executes each of the five framework activities in sequence, beginning with communication and culminating with deployment.
- An iterative process flow repeats one or more of the activities before proceeding to the next.
- An evolutionary process flow executes the activities in a “circular” manner. Each circuit through the five activities leads to a more complete version of the software.
- A parallel process flow executes one or more activities in parallel with other activities.

(3) Umbrella Activities

- Software engineering process framework activities are complemented by a number of umbrella activities.
- In general, umbrella activities are applied throughout a software project and help a software team manage and control progress, quality, change, and risk.

Typical umbrella activities include:

Software project tracking and control—allows the software team to assess progress against the project plan and take any necessary action to maintain the schedule.

Risk management—assesses risks that may affect the outcome of the project or the quality of the product.

Software quality assurance—defines and conducts the activities required to 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.

(4) Prescriptive Process Models

- Prescriptive process models were originally proposed to bring order to the chaos of software development.
- Traditional models have brought a certain amount of useful structure to software engineering work and have provided a reasonably effective road map for software teams.
- Following are prescriptive process models:
  - Waterfall model
  - Incremental Model
    - RAD Model
  - Evolutionary Process Model
    - Prototype Model
    - Spiral Model
    - Concurrent Process Model

(5) Waterfall Process 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, culminating in ongoing support of the completed software.

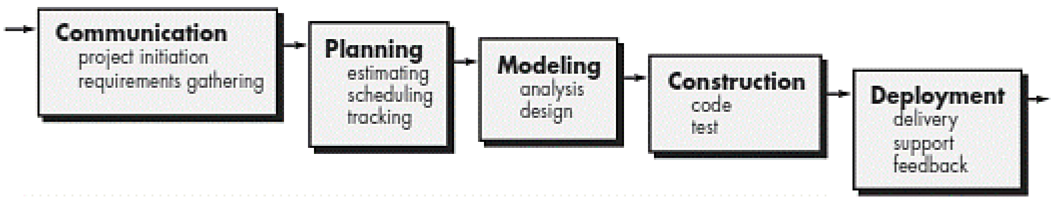


Figure: Waterfall Model

Limitations:

- The nature of the requirements will not change very much during development; during evolution.
- The model implies that you should attempt to complete a given stage before moving on to the next stage.
- Does not account for the fact that requirements constantly change.
- It also means that customers cannot use anything until the entire system is complete.
- The model implies that once the product is finished, everything else is maintenance.
- Surprises at the end are very expensive
- Some teams sit ideal for other teams to finish
- Therefore, this model is only appropriate when the requirements are well-understood and changes will be fairly limited during the design process.

When to use waterfall model?

- Requirements are very well known, clear and fixed
- Product definition is stable
- Technology is understood
- There are no ambiguous requirements
- Ample resources with required expertise are available freely
- The project is short

(6) Incremental Process Model

- The incremental model combines elements of linear and parallel process flows.
- The incremental model applies linear sequences in a staggered fashion as calendar time progresses.
- Each linear sequence produces deliverable "increments" of the software.
- For example, word-processing software developed using the incremental paradigm might deliver basic file management, editing, and document production functions in the first increment; more sophisticated editing and document production capabilities in the second increment; spelling and grammar checking in the third increment; and advanced page layout capability in the fourth increment.

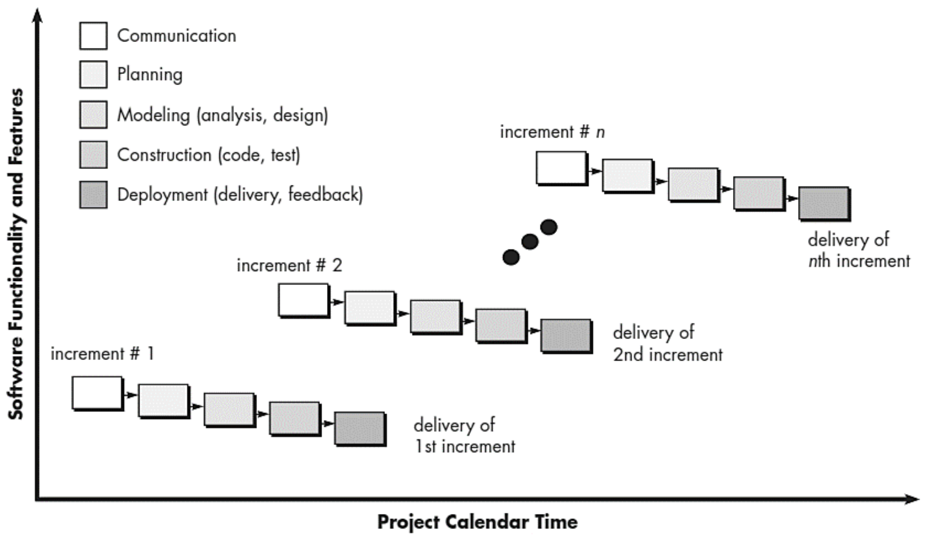

Communication
Planning
Modeling (analysis, design)
Construction (code, test)
Deployment (delivery, feedback)
increment # n
delivery of nth increment
increment # 2
delivery of 2nd increment
increment # 1
delivery of 1st increment
Software Functionality and Features
Project Calendar Time

Figure: Incremental Process Model

Advantages:

- Generates working software quickly and early during the software life cycle.
- This model is more flexible – less costly to change scope and requirements.
- It is easier to test and debug during a smaller iteration.
- In this model customer can respond to each built.
- Lowers initial delivery cost.
- Easier to manage risk because risky pieces are identified and handled during iteration.

Disadvantages:

- Needs good planning and design.
- Needs a clear and complete definition of the whole system before it can be broken down and built incrementally.
- Total cost is higher than waterfall.

When to use waterfall model?

- This model can be used when the requirements of the complete system are clearly defined and understood.
- Major requirements must be defined; however, some details can evolve with time.
- There is a need to get a product to the market early.
- A new technology is being used.
- Resources with needed skill set are not available.
- There are some high risk features and goals.

(7) RAD (Rapid Application Development) Process Model

- It is a type of incremental model. In RAD model the components or functions are developed in parallel as if they were mini projects.
- The developments are time boxed, delivered and then assembled into a working prototype.
- This can quickly give the customer something to see and use and to provide feedback regarding the delivery and their requirements.

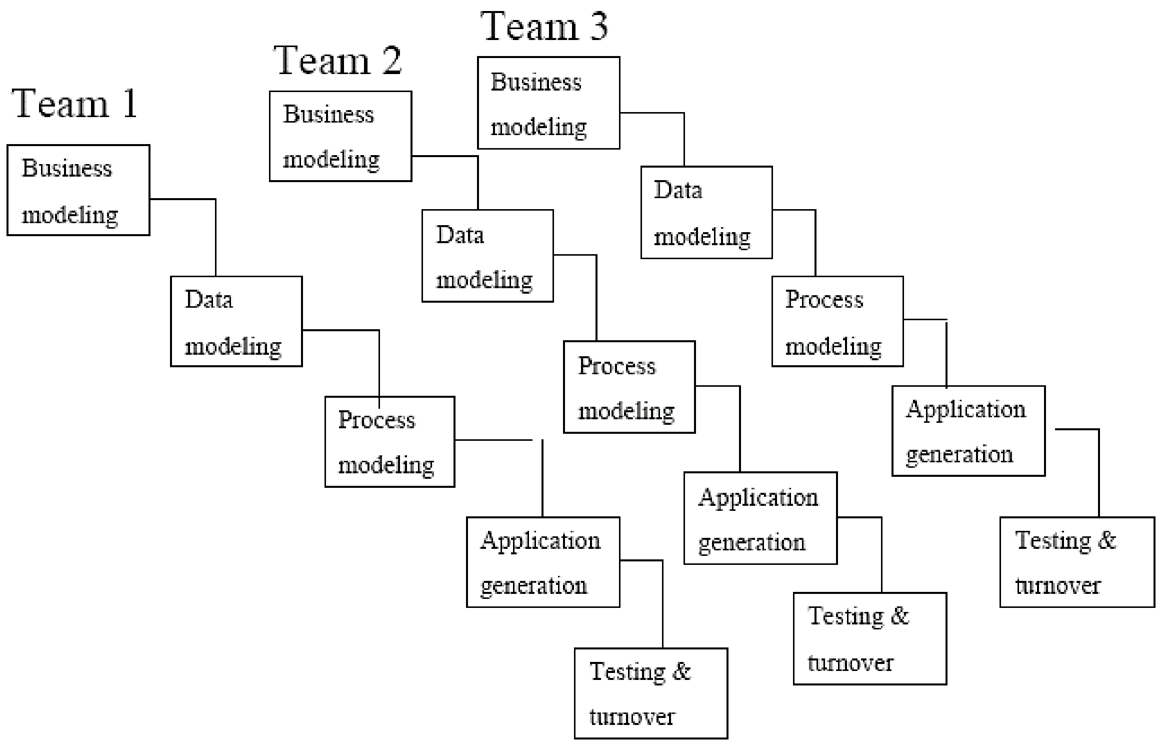


Figure: Rapid Application Development Model

Advantages:

- Reduced development time.
- Increases reusability of components
- Quick initial reviews occur
- Encourages customer feedback
- Integration from very beginning solves a lot of integration issues.

Disadvantages:

- For large but scalable projects RAD requires sufficient human resources.
- Projects fail if developers and customers are not committed in a much shortened time-frame.
- Problematic if system cannot be modularized.
- Not appropriate when technical risks are high (heavy use of new technology).

When to use RAD model?

- RAD should be used when there is a need to create a system that can be modularized in 2-3 months of time.
- It should be used if there’s high availability of designers for modeling and the budget is high enough to afford their cost along with the cost of automated code generating tools.
- RAD SDLC model should be chosen only if resources with high business knowledge are available and there is a need to produce the system in a short span of time (2-3 months).

(8) Prototype process model

- The basic idea here is that instead of freezing the requirements before a design or coding can proceed, a throwaway prototype is built to understand the requirements.
- This prototype is developed based on the currently known requirements.
- By using this prototype, the client can get an "actual feel" of the system, since the interactions with prototype can enable the client to better understand the requirements of the desired system.
- Prototyping is an attractive idea for complicated and large systems for which there is no manual process or existing system to help determining the requirements.
- The prototype are usually not complete systems and many of the details are not built in the prototype. The goal is to provide a system with overall functionality.

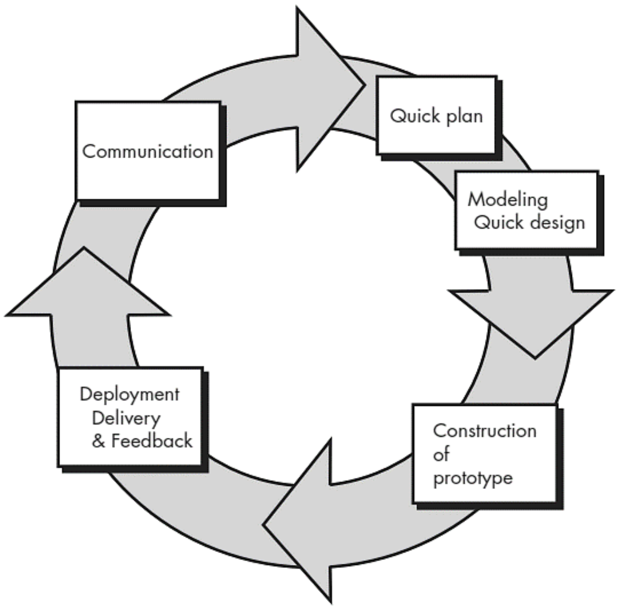


Figure: Prototype Model

Advantages:

- Users are actively involved in the development
- Since in this methodology a working model of the system is provided, the users get a better understanding of the system being developed.
- Errors can be detected much earlier.
- Quicker user feedback is available leading to better solutions.
- Missing functionality can be identified easily
- Confusing or difficult functions can be identified

Disadvantages:

- Leads to implementing and then repairing way of building systems.
- Practically, this methodology may increase the complexity of the system as scope of the system may expand beyond original plans.
- Incomplete application may cause application not to be used as the full system was designed.

When to use Prototype Model?

- Prototype model should be used when the desired system needs to have a lot of interaction with the end users.
- Typically, online systems, web interfaces have a very high amount of interaction with end users, are best suited for Prototype model. It might take a while for a system to be built that allows ease of use and needs minimal training for the end user.
- Prototyping ensures that the end users constantly work with the system and provide a feedback which is incorporated in the prototype to result in a useable system. They are excellent for designing good human computer interface systems.

(9) Spiral process model

- It was originally proposed by Barry Boehm, the spiral model is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model.
- It provides the potential for rapid development of increasingly more complete versions of the software.
- The spiral development model is a risk-driven process model generator that is used to guide multi-stakeholder concurrent engineering of software intensive systems.
- It has two main distinguishing features. One is a cyclic approach for incrementally growing a system's degree of definition and implementation while decreasing its degree of risk.
- The other is a set of anchor point milestones for ensuring stakeholder commitment to feasible and mutually satisfactory system solutions.

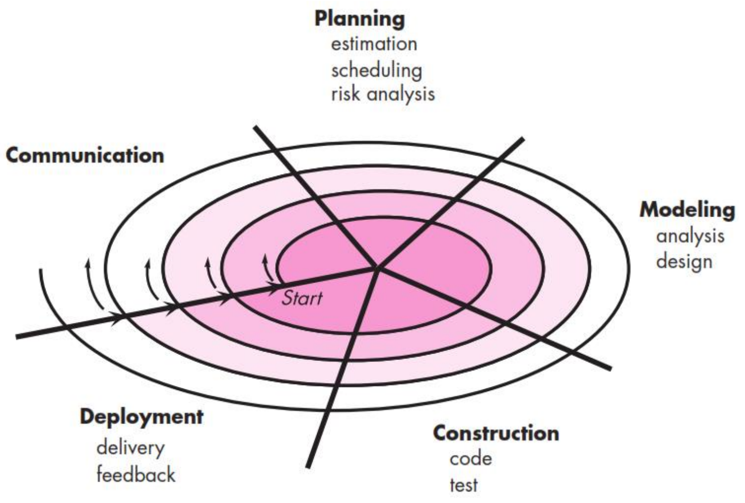


Figure: Spiral Process Model

- The first circuit around the spiral might result in the development of a product specification; subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software.
- Each pass through the planning region results in adjustments to the project plan.
- Cost and schedule are adjusted based on feedback derived from the customer after delivery.
- In addition, the project manager adjusts the planned number of iterations required to complete the software.

Advantages:

- High amount of risk analysis hence, avoidance of Risk is enhanced.
- Good for large and mission-critical projects.
- Strong approval and documentation control.
- Additional Functionality can be added at a later date.
- Software is produced early in the software life cycle.

Disadvantages:

- Can be a costly model to use.
- Risk analysis requires highly specific expertise.
- Project's success is highly dependent on the risk analysis phase.
- Doesn't work well for smaller projects.

When to use spiral process model?

- When costs and risk evaluation is important for medium to high-risk projects.
- Long-term project commitment unwise because of potential changes to economic priorities
- Users are unsure of their needs.
- Requirements are complex
- New product line Significant changes are expected (research and exploration)

(10) Concurrent Process Model

- The concurrent development model, sometimes called concurrent engineering, allows a software team to represent iterative and concurrent elements of any of the process models.
- For example, the modeling activity defined for the spiral model is accomplished by invoking one or more of the following software engineering actions: prototyping, analysis, and design.
- The activity—modeling—may be in any one of the states noted at any given time.
- Similarly, other activities, actions, or tasks (e.g., communication or construction) can be represented in an analogous manner.
- All software engineering activities exist concurrently but reside in different states.

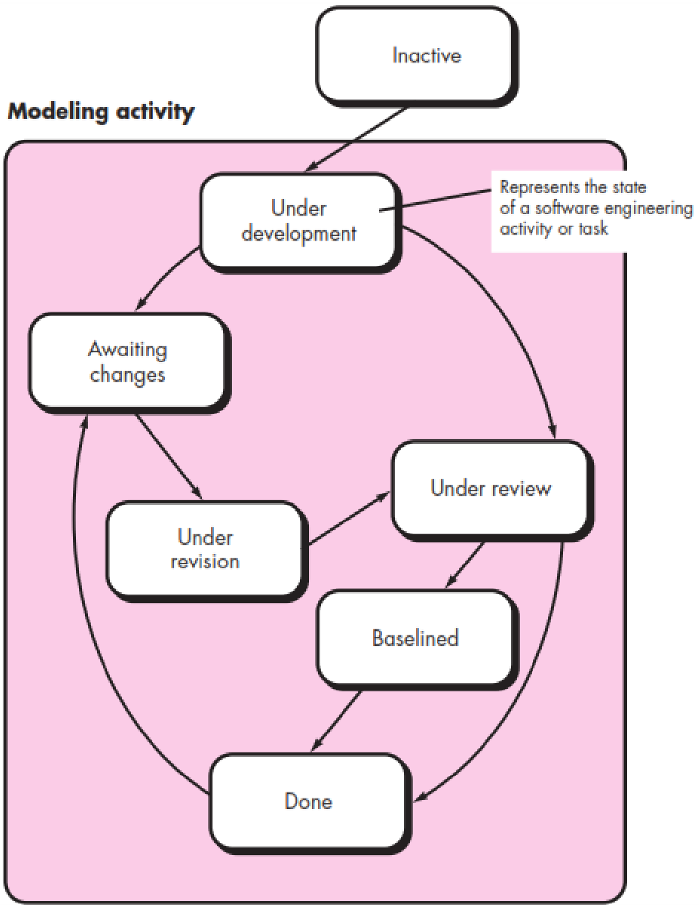


Figure: Concurrent Process Model

(11) Component Based Development

- Commercial off-the-shelf (COTS) software components, developed by vendors who offer them as products, provide targeted functionality with well-defined interfaces that enable the component to be integrated into the software that is to be built.
- The component-based development model incorporates many of the characteristics of the spiral model.
- It is evolutionary in nature, demanding an iterative approach to the creation of software.
- However, the component-based development model constructs applications from prepackaged software components.

The component-based development model incorporates the following steps:

1. Available component-based products are researched and evaluated for the application domain in question.
2. Component integration issues are considered.
3. A software architecture is designed to accommodate the components.
4. Components are integrated into the architecture.
5. Comprehensive testing is conducted to ensure proper functionality.

- The component-based development model leads to software reuse, and reusability provides software engineers with a number of measurable benefits.

(12) Agile Process Model

- Agile SDLC model is a combination of iterative and incremental process models with focus on process adaptability and customer satisfaction by rapid delivery of working software product.
- Agile Methods break the product into small incremental builds.
- Every iteration involves cross functional teams working simultaneously on various areas like planning, requirements analysis, design, coding, unit testing, and acceptance testing.

Advantages:

- Customer satisfaction by rapid, continuous delivery of useful software.
- Customers, developers and testers constantly interact with each other.
- Close, daily cooperation between business people and developers.
- Continuous attention to technical excellence and good design.
- Regular adaptation to changing circumstances.
- Even late changes in requirements are welcomed

Disadvantages:

- In case of some software, it is difficult to assess the effort required at the beginning of the software development life cycle.
- There is lack of emphasis on necessary designing and documentation.
- The project can easily get taken off track if the customer representative is not clear what final outcome that they want.
- Only senior programmers are capable of taking the kind of decisions required during the development process.

(13) Product and Process

- If the process is weak, the end product will undoubtedly suffer.
- But a compulsive overreliance on process is also dangerous.
- People derive as much (or more) satisfaction from the creative process as they do from the end product.
- An artist enjoys the brush strokes as much as the framed result.
- A writer enjoys the search for the proper metaphor as much as the finished book.
- As creative software professional, you should also derive as much satisfaction from the process as the end product.
- The duality of product and process is one important element in keeping creative people engaged as software engineering continues to evolve.

(1) Agility and Agile Process Model

Agility:

- Agility is dynamic, content specific, aggressively change embracing, and growth oriented
- It encourages team structures and attitudes that make communication (among team members, between technologists and business people, between software engineers and their managers) more simplistic.
- It emphasizes rapid delivery of operational software and de-emphasizes the importance of intermediate work products.
- It recognizes that planning in an uncertain world has its limits and that a project plan must be flexible.
- Agility can be applied to any software process.

Agile Process:

- Any agile software process is characterized in a manner that addresses a number of key assumptions

  [1] It is difficult to predict in advance which software requirements will persist and which will change. It is equally difficult to predict how customer priorities will change as the project proceeds.

  [2] For many types of software, design and construction are interleaved. That is, both activities should be performed in tandem so that design models are proven as they are created. It is difficult to predict how much design is necessary before construction is used to prove the design.

  [3] Analysis, design, construction, and testing are not as predictable as we might like.

Agility Principles:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity—the art of maximizing the amount of work not done—is essential.
11. The best architectures, requirements, and designs emerge from self– organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Agile Process Models:

- Extreme Programming (XP)
- Adaptive Software Development (ASD)
- Dynamic Systems Development Method (DSDM)
- Scrum
- Crystal
- Feature Driven Development (FDD)
- Agile Modeling (AM)

(2) Extreme Programming

- It is most widely used agile process model.
- XP uses an object-oriented approach as its preferred development paradigm.
- It defines four (4) framework activities **Planning, Design, Coding, and Testing.**

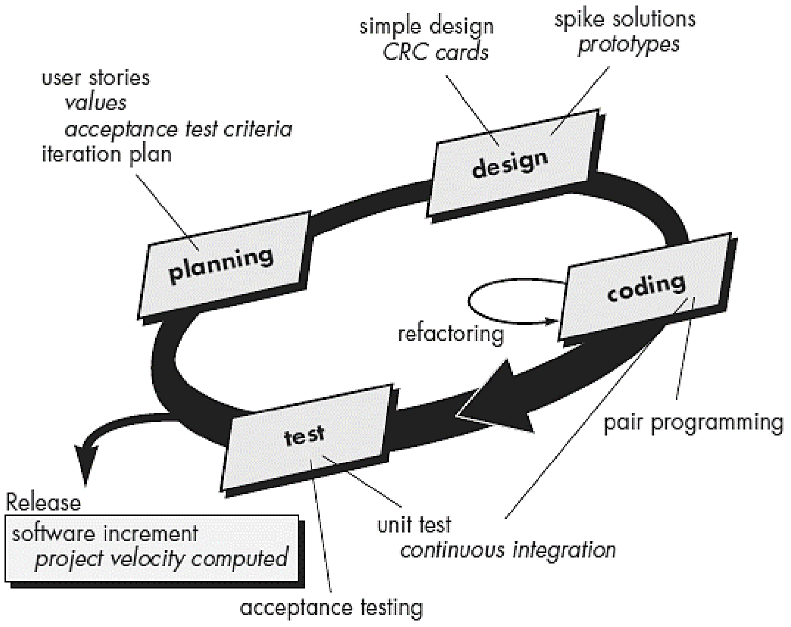


Figure: Extreme Programming Process

Planning:

- Begins with the creation of a set of stories (also called user stories)
- Each story is written by the customer and is placed on an index card
- The customer assigns a value (i.e. a priority) to the story
- Agile team assesses each story and assigns a cost
- Stories are grouped to for a deliverable increment
- A commitment is made on delivery date
- After the first increment "project velocity" is used to help define subsequent delivery dates for other increments

Design:

- Follows the 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
- Encourages "refactoring"—an iterative refinement of the internal program design
- Design occurs both before and after coding commences

Coding:

- Recommends the construction of a series of unit tests for each of the stories before coding commences
- Encourages "pair programming"
    - Developers work in pairs, checking each other's work and providing the support to always do a good job.
    - 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

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

Adaptive Software Development (ASD)

Adaptive Software Development (ASD) is a technique for building complex software and systems.
ASD focus on human collaboration and team self-organization.
ASD incorporates three phases **Speculation, Collaboration, and Learning.**

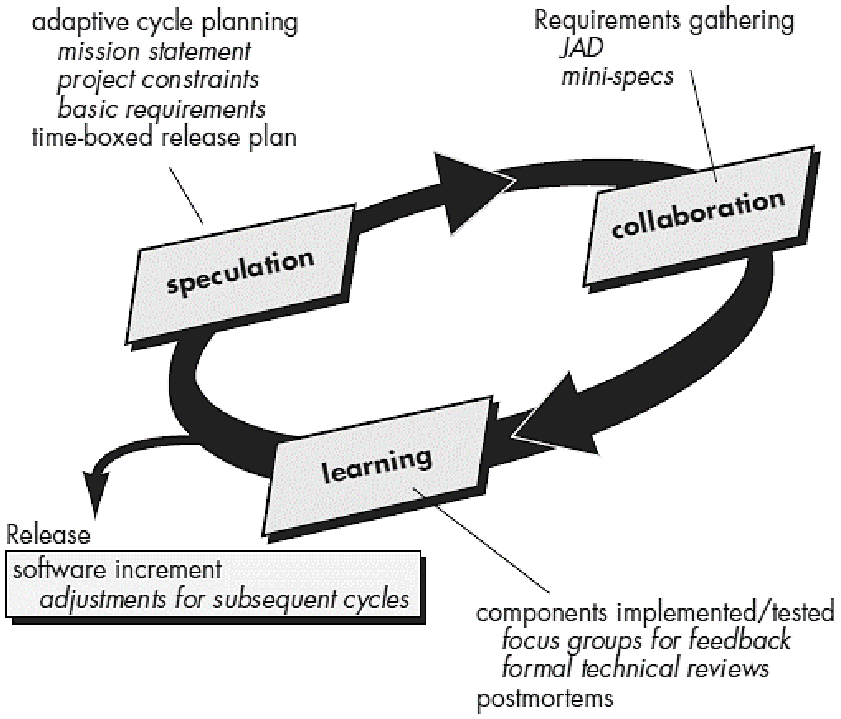


Figure: Adaptive Software Development

Speculation:

- "Speculate" refers to the planning paradox—outcomes are unpredictable, therefore, endless suppositions on a product's look and feel are not likely to lead to any business value.
- The big idea behind speculate is when we plan a product to its smallest detail as in a requirements up front Waterfall variant, we produce the product we intend and not the product the customer needs.
- In the ASD mindset, planning is to speculation as intention is to need.

Collaboration:

- Collaboration represents a balance between managing the doing and creating and maintaining the collaborative environment."

- Speculation says we can’t predict outcomes. If we can’t predict outcomes, we can’t plan. If we can’t plan, traditional project management theory suffers.
- Collaboration weights speculation in that a project manager plans the work between the predictable parts of the environment and adapts to the uncertainties of various factors—stakeholders, requirements, software vendors, technology, etc.

Learning:

- “Learning” cycles challenge all stakeholders and project team members.
- Based on short iterations of design, build and testing, knowledge accumulates from the small mistakes we make due to false assumptions, poorly stated or ambiguous requirements or misunderstanding the stakeholders’ needs.
- Correcting those mistakes through shared learning cycles leads to greater positive experience and eventual mastery of the problem domain.

Dynamic Systems Development Methods (DSDM)

- The Dynamic Systems Development Method is an agile software 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”
- DSDM is an iterative software process in which each iteration follows the 80 percent rule.
- That is, only enough work is required for each increment to facilitate movement to the next increment.
- The remaining detail can be completed later when more business requirements are known or changes have been requested and accommodated.

DSDM life cycle that defines three different iterative cycles, preceded by two additional life cycle activities:

Feasibility study—establishes the basic business requirements and constraints associated with the application to be built and then assesses whether the application is a viable candidate for the DSDM process.

Business study—establishes the functional and information requirements that will allow the application to provide business value; also, defines the basic application architecture and identifies the maintainability requirements for the application.

Functional model iteration—produces a set of incremental prototypes that demonstrate functionality for the customer.

Design and build iteration—revisits prototypes built during functional model iteration to ensure that each has been engineered in a manner that will enable it to provide operational business value for end users.

Implementation—places the latest software increment into the operational environment.

- DSDM can be combined with XP to provide a combination approach that defines a solid process model (the DSDM life cycle) with the nuts and bolts practices (XP) that are required to build software increments.
- In addition, the ASD concepts of collaboration and self-organizing teams can be adapted to a combined process model.

Scrum

- Scrum principles are consistent with the agile manifesto and are used to guide development activities within a process that incorporates the five framework activities: **requirements, analysis, design, evolution, and delivery**.
- Within each framework activity, work tasks occur within a process pattern called a sprint.

- The work conducted within a sprint (the number of sprints required for each framework activity will vary depending on product complexity and size) is adapted to the problem at hand and is defined and often modified in real time by the Scrum team.
- Scrum emphasizes the use of a set of software process patterns that have proven effective for projects with tight timelines, changing requirements, and business criticality.
- Each of these process patterns defines a set of development actions: Backlog—a prioritized list of project requirements or features that provide business value for the customer.
- Items can be added to the backlog at any time (this is how changes are introduced).
- The product manager assesses the backlog and updates priorities as required.

Crystal

- The Crystal methodology is one of the most lightweight, adaptable approaches to software development. Crystal is actually comprised of a family of agile methodologies such as Crystal Clear, Crystal Yellow, Crystal Orange and others, whose unique characteristics are driven by several factors such as team size, system criticality, and project priorities.
- This Crystal family addresses the realization that each project may require a slightly tailored set of policies, practices, and processes in order to meet the project‘s unique characteristics.
- Several of the key tenets of Crystal include teamwork, communication, and simplicity, as well as reflection to frequently adjust and improve the process.
- Like other agile process methodologies, Crystal promotes early, frequent delivery of working software, high user involvement, adaptability, and the removal of bureaucracy or distractions.

Feature Driven Development (FDD)

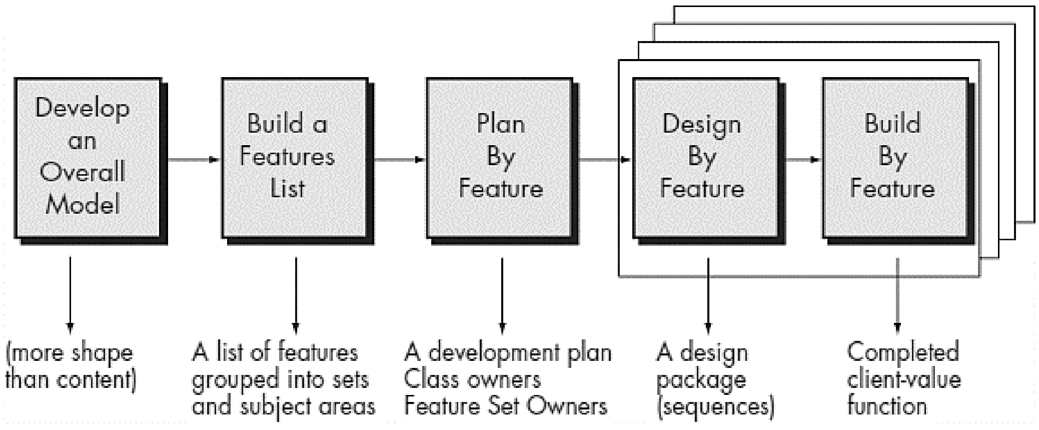

Develop an Overall Model
Build a Features List
Plan By Feature
Design By Feature
Build By Feature
(more shape than content)
A list of features grouped into sets and subject areas
A development plan Class owners Feature Set Owners
A design package (sequences)
Completed client-value function

Figure: Feature Driven Development Model

- FDD is a model-driven, short-iteration process.
- It begins with establishing an overall model shape.
- Then it continues with a series of two-week “design by feature, build by feature” iterations.
- The features are small, “useful in the eyes of the client” results.
- FDD designs the rest of the development process around feature delivery using the following eight practices:
  - Domain Object Modeling
  - Developing by Feature
  - Component/Class Ownership