CHAPTER - 1

Introduction to Software Engineering

1.1. Computer software

Computer software is the product that software professionals build and then support over the long term. A textbook description of software might take the following form:

Software is:

1. Instructions (computer programs) that when executed provide desired features, function, and performance;
2. Data structures that enable the programs to adequately manipulate information, and
3. Descriptive information in both hard copy and virtual forms that describes the operation and use of the programs.

Software has characteristics that are considerably different than those of hardware:

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

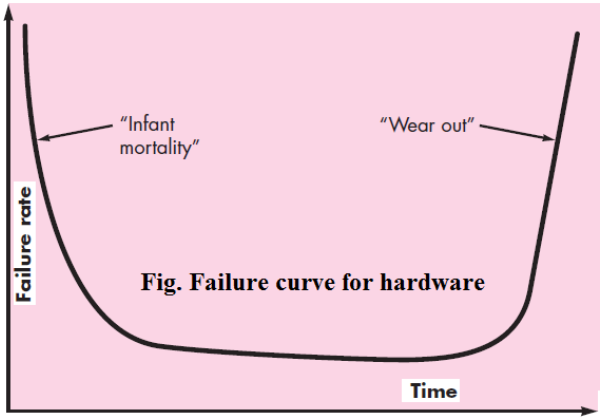
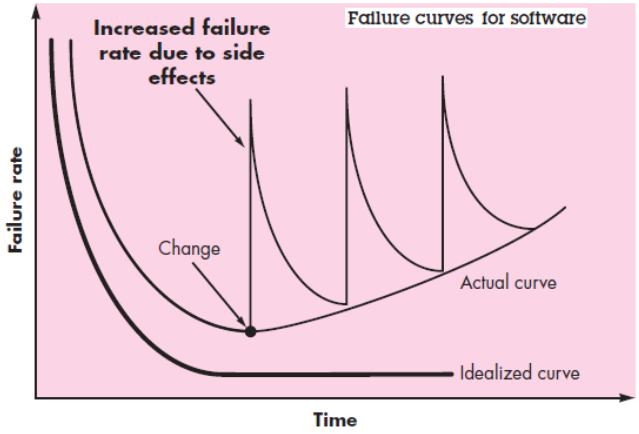


Fig. Failure curve for hardware



1.2 History of Software Engineering

- In the 1950's software development was de-emphasized, because it only contributed to about 20% of overall system cost. The hardware manufacturing company generally assign their software job to third party.
- Programmers moved from machine language, to assembly language, to high-level language. Also, the concern of organizational privacy increase and the hardware manufacturing company stated to develop software for their hardware product.
- In 1968, a NATO report coined the term "software engineering". It discuss about the group work for software engineering in more systematic way.
- Hardware became faster and cheaper, outpacing the ability of software to keep up
- By the 1980's the software cost of a system had risen to 80%, and many experts pronounced the field "in crisis" because the software development face the problem of not delivered on time, over budget, unmaintainable due to poor design and documentation, and unreliable due to poor system analysis and testing. Increase the use of object-oriented programming through languages such as C++ and Objective-C.
- Java is developed and released in the mid-1990s. Increasing attention paid to notions of software architecture. Client-server distributed architectures are increasingly used. The UML is proposed.
- Today, the use of integrated development environments becomes more common. Use of stand-alone CASE tools declines. Use of the UML becomes widespread. Increasing use of scripting languages such as Python and PERL for software development. C# developed as a competitor to Java.

1.3 Types of Software

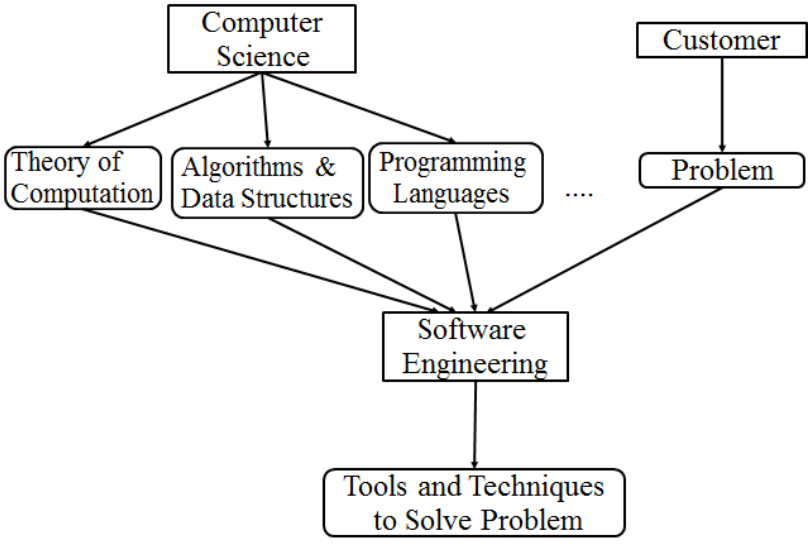
Software is the collection of computer programs, procedures and documentation that performs different tasks on a computer system. The term 'software' was first used by John Tukey in 1958. At the very basic level, computer software consists of a machine language that comprises groups of binary values, which specify processor instructions. The processor instructions change the state of computer hardware in a predefined sequence. Briefly, computer software is the language in which a computer speaks. Computer software can be put into categories based on common function, type, or field of use. There are three broad classifications:

- **Application Software:** It enables the end users to accomplish certain specific tasks. They include programs such as web browsers, office software, games, and programming tools.

- **System Software:** It helps in running computer hardware and the computer system. System software refers to the operating systems; device drivers, servers, windowing systems and utilities (e.g. anti-virus software, firewalls, disk defragmenters).

- **Malicious Software or Malware:** Malware refers to any malicious software to harm and disrupt computers. Adware, spyware, computer viruses, worms, Trojan horses and scareware are malware.

1.4 Software Engineering



Software engineering encompasses a process, a collection of methods (practice) and an array of tools that allow professionals to build high quality computer software in a timely manner. Thus, the software engineering includes overall activities performed during the production of software such as requirement analysis, planning, designing, estimation, scheduling and module development, combination of modules for overall product, testing the deliverables, delivery to the owner, and finally feedback from the customers. Now, the same cycle repeats again in response to the feedback for making the software product more energetic and more prefect as the user requirement.

The main remarkable features of software engineering are listed as:

- Highly innovative and rapidly changing field
- Few results are supported by empirical or comparative studies;
- Work within the field older than 3–4 years is rarely acknowledged or referenced;
- Old problems are given new names and old solutions overlooked;
- There is a need for interdisciplinary work comprising e.g. mathematics, psychology, business or management science, etc.

The Role of Software Engineering in System Design

Software is what enables us to use computer hardware effectively and is needed for our modern life. A software system is often a component of a much larger system. The software engineering activity is therefore a part of a much larger system design activity in which the requirements of the software are balanced against the requirements of other parts of system being designed e.g. a telephone switching system consists of computers, telephone lines and cables, telephones, satellites and finally software to control the various other components. It is the combination of all these components that is expected to meet the requirements of the whole system. Power plant or traffic monitoring system, banking system, hospital administration system are other examples of systems that exhibit the need to view the software as a component of a larger system.

A. The software process

When you walk to build a product or system, it is important to go through a series of predictable steps: a road map that helps you to create a timely, high quality result. The road map that you follow is called a "Software Process".

Thus, a *process* is a collection of activities, actions, and tasks that are performed when some work product is to be created. In the context of software engineering, a process is *not* a rigid prescription for how to build computer software. Rather, it is an adaptable approach that enables the people doing the work (the software team) to pick and choose the appropriate set of work actions and tasks. The intent is always to deliver software in a timely manner and with sufficient quality to satisfy those who have sponsored its creation and those who will use it.

B. The software process Framework

A *process framework* establishes the foundation for a complete software engineering process by identifying a small number of *framework activities* that are applicable to all software projects, regardless of their size or complexity. In addition, the process framework encompasses a set of *umbrella activities* that are applicable across the entire software process. A generic process framework for software engineering encompasses five activities:

i. Communication

Before any technical work can commence, it is critically important to communicate and collaborate with the customer and other stakeholders. A *stakeholder* is anyone who has a stake in the successful outcome of the project—business managers, end users, software engineers, support people, etc. The intent is to understand stakeholders' objectives for the project and to gather requirements that help define software features and functions.

ii. Planning

As, any complicated journey can be simplified if a *map* exists, a software project requires *software project plan* that defines the software engineering work by describing the technical tasks to be conducted, the risks that are likely, the resources that will be required, the work products to be produced, and a work schedule.

iii. Modeling

As a bridge builder, or an architect, or carpenter etc. work with models every day to understand the big picture about product such as what it will look like architecturally, how the constituent parts fit together, greater detail in an effort to better understand the problem and how you're going to solve it. A software engineer does the same thing by creating models to better understand software requirements and the design that will achieve those requirements.

iv. Construction

This activity combines code generation (either manual or automated) and the testing that is required uncovering errors in the code.

v. Deployment

The software (as a complete entity or as a partially completed increment) is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation.

The details of the software process will be quite different in each case, but the framework activities remain the same. For many software projects, framework activities are applied iteratively as a project progresses. Each project iteration produces a *software increment* that provides stakeholders with a subset of overall software features and functionality. As each increment is produced, the software becomes more and more complete. The iterative process models are the prototype model, spiral model etc. Now, the question is: Who create software process model? The answer is: The software engineers and their managers adopt the process to their needs and then follow it. In addition, the people who have requested the software have a role to play in process of defining, building, and testing it.

C. The 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
- Risk management
- Software quality assurance
- Technical reviews
- Measurement
- Software configuration management
- Reusability management
- Work product preparation and production

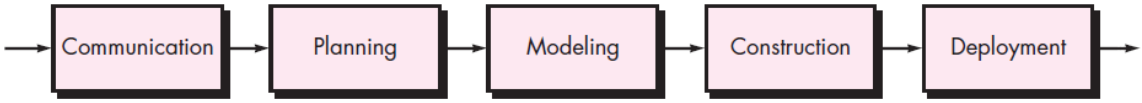
Thus, the umbrella activities occur throughout the software process to their need and then follow it. In addition, the people who have requested the software help to management, tracking, and control the software as umbrella activities.

D. The Generic Process Model

A generic process framework for software engineering defines five framework activities—communication, planning, modeling, construction, and deployment. In addition, a set of umbrella activities are applied throughout the process. Now, one important aspect of the software process has not yet been discussed, called *process flow* that 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

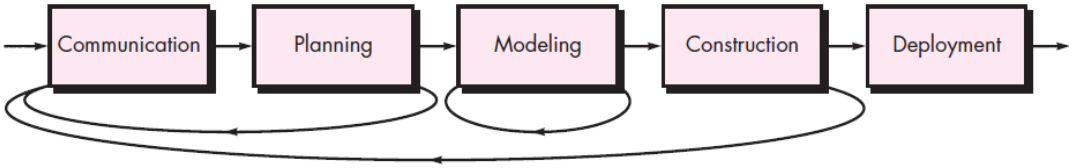
A *linear process flow* executes each of the five framework activities in sequence, beginning with communication and culminating with deployment.



(a) Linear process flow

(b) Iterative Process Flow

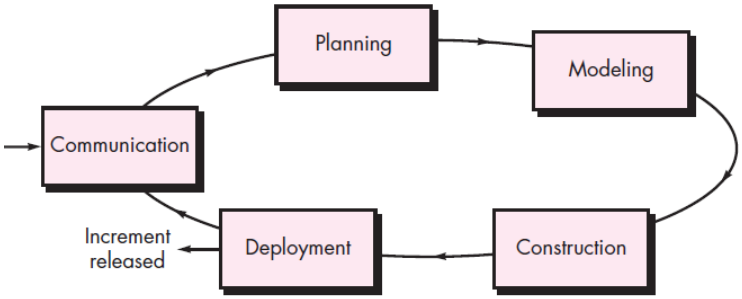
An *iterative process flow* repeats one or more of the activities before proceeding to the next.



(b) Iterative process flow

(c) Evolutionary Process Flow

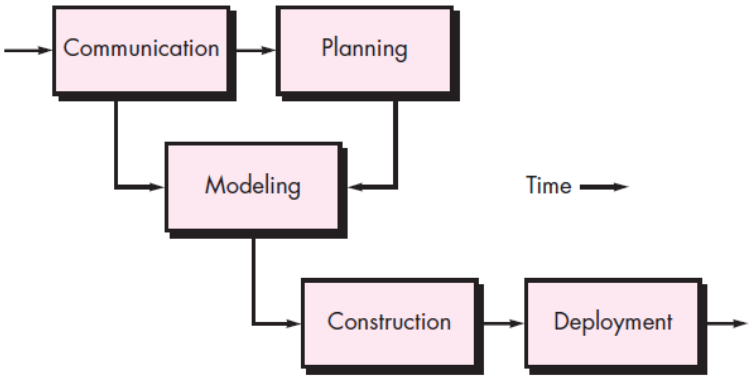
An *evolutionary process flow* executes the activities in a "circular" manner. Each circuit through the five activities leads to a more complete version of the software.



(c) Evolutionary process flow

(d) Parallel Process Flow

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



(d) Parallel process flow

1.5 Perspective Process Models

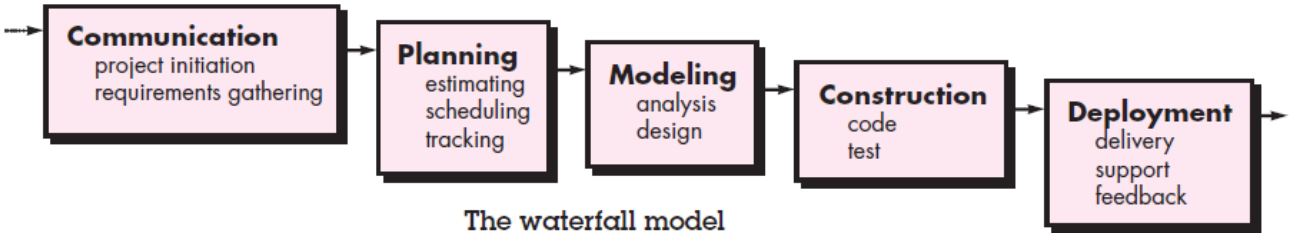
The software development life cycle (SDLC) illustrate us the general prescriptive process model for the development of software product. Prescriptive (Decided) or traditional process models were originally proposed to bring order to the vague of software development. They are called "Prescriptive" because they prescribe a set of process elements such as framework activities, software engineering actions, tasks, and work products, quality assurance, and change control mechanism for each project. History indicates that these traditional models have brought a certain amount of useful structure to software engineering work & have provided a reasonably effective road map for software team.

All software process models can accommodate the generic framework activities but each applies a different emphasis to these activities and defines a process flow that invokes each framework activity (as well as software engineering actions and tasks) in a different manner. The generic type of software process models are:

▪ Linear Sequential Model (Water Fall Model) ▪ Prototyping Model ▪ RAD Model ▪ Evolutionary Model ▪ Incremental Model ▪ Spiral Model	▪ Win-Win Spiral Model ▪ Concurrent Development Model ▪ Component Based Development ▪ Formal Methods Models ▪ Fourth Generation Technology Models

a. The Waterfall Model

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



The waterfall model

The linear process flow model will be appropriate when:

- User requirements are clearly defined
- Less number of staffing to do simultaneous work on that project in the organization
- No much re-useable components
- Sufficient time for software development
- Less communication between customer and developer
- Software engineer have clear idea to do the prescribed task

The V-model

A variation in the representation of the waterfall model is called the *V-model*. The V-model provides a way of visualizing how verification and validation actions are applied to earlier engineering work. The V-model depicts the relationship of quality assurance actions to the actions associated with communication, modeling, and early construction activities. As a software team moves down the left side of the V, basic problem requirements are refined into progressively more detailed and technical representations of the problem and its solution. Once code has been generated, the team moves up the right side of the V, essentially performing a series of tests (quality assurance actions) that validate each of the models created as the team moved down the left side

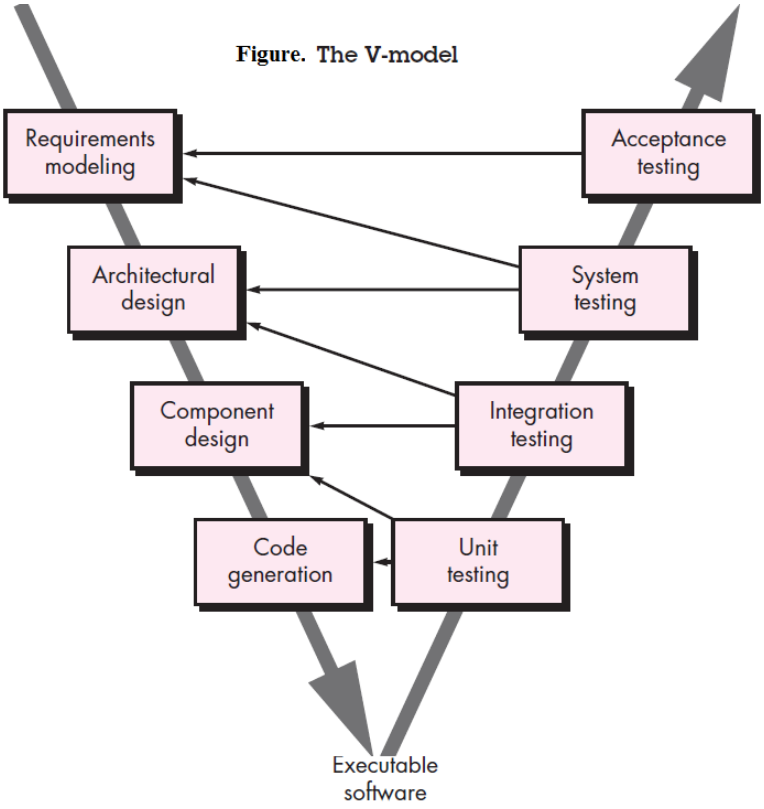
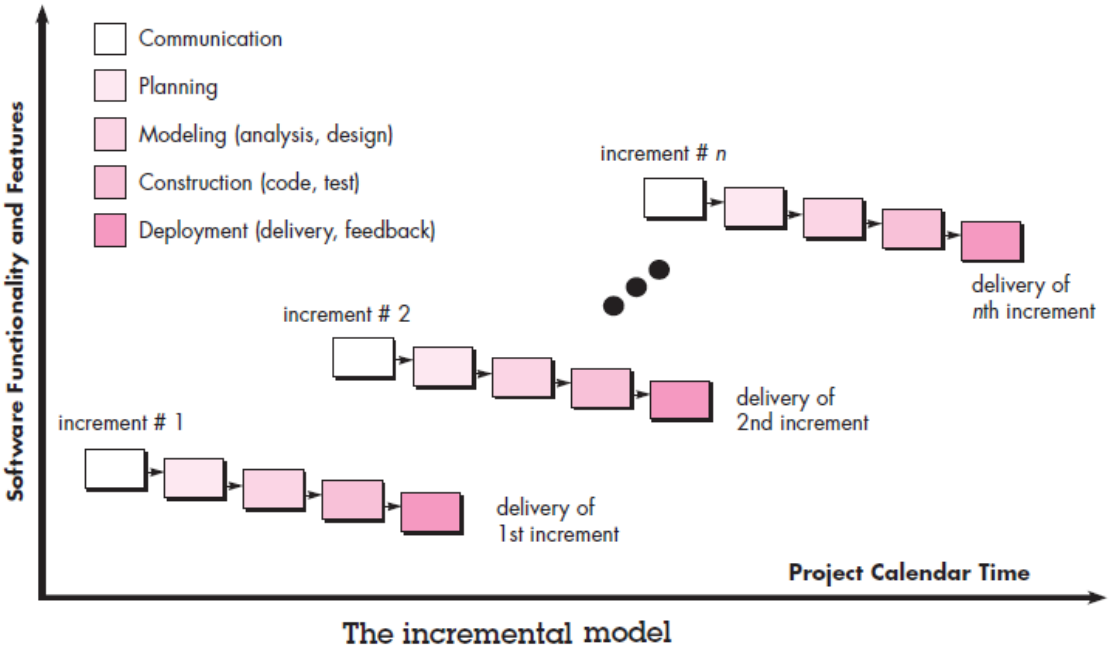


Figure. The V-model

b. Rapid Application Development (RAD) Model

The *incremental* model or *Rapid application development* (RAD) model combines elements of linear and parallel process flows discussed. Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project. If the core product is well received, then additional staff (if required) can be added to implement the next increment. In addition, increments can be planned to manage technical risks.



The incremental model

The RAD model will be appropriate when:

i. This is appropriate both for less staffing (manually work on small module) or large staffing (parallel work on project).
ii. We have to develop the application in short period of time.
iii. Customer clearly defined their requirements.
iv. No much re-useable components or no chances of outsourcing in organization.

c. **Evolutionary Process Models**

Evolutionary models are iterative which is much suitable for new systems where no clear idea of the requirements, inputs and outputs parameters. It produces an increasingly more complete version of the software with each iteration. The evolutionary process models are appropriate in following situation:

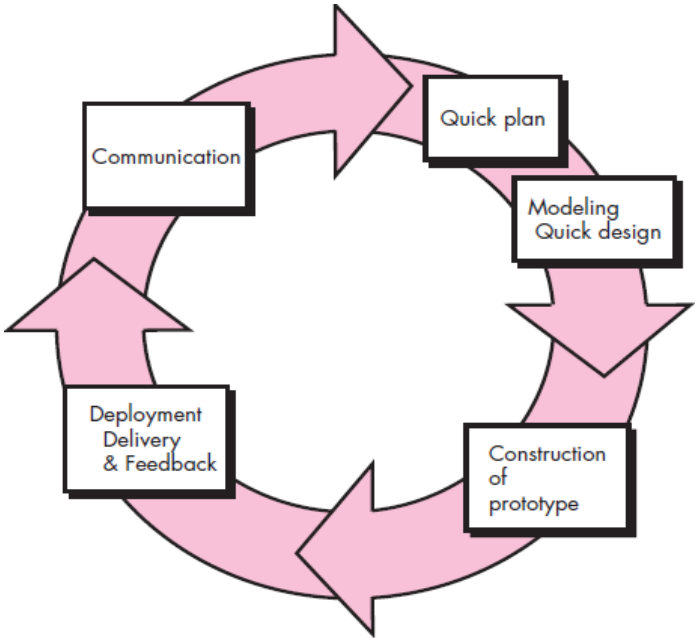
- No clear idea about the product for both customers and developers.
- Good communication between the customers and developers.
- Sufficient time for the software development
- Less chances of outsourcing and availability of re-usable components.

The two common evolutionary process models are:

i. Prototyping

Although prototyping can be used as a stand-alone process model, it assists you and other stakeholders to better understand what is to be built when requirements are fuzzy. The prototyping paradigm begins with communication to stakeholders. Then, prototyping iteration is planned quickly, and modeling occurs. The quick design leads to the construction of a prototype. The prototype is deployed and evaluated by stakeholders, who provide feedback that is used to further refine requirements.

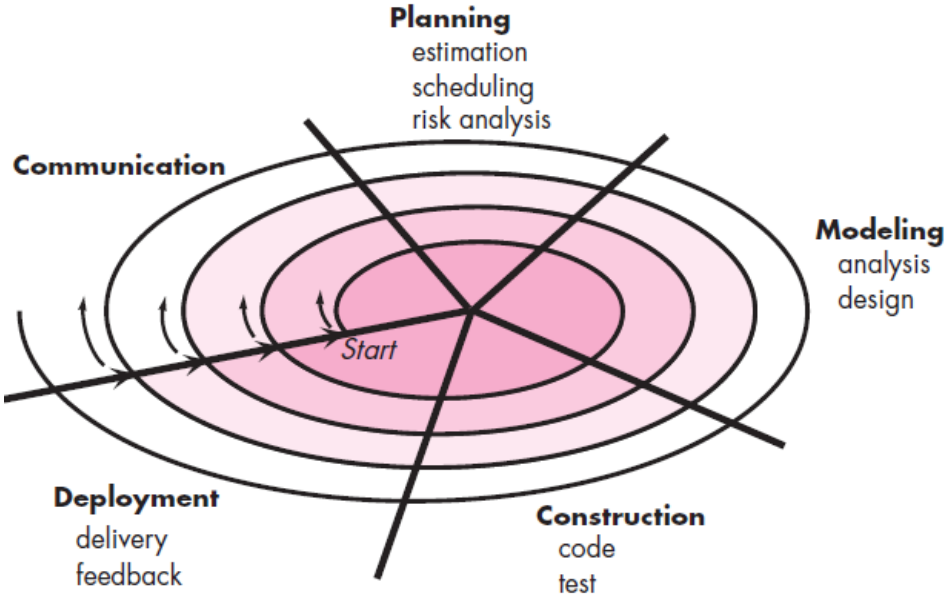
Based upon the user feedback the, requirements are re-defined and this cycle continue until the user accepts the prototype. The actual system is then developed using the classical waterfall approach or may be other prescriptive model.



The prototyping paradigm

ii. The Spiral Model.

The *spiral model* was originally proposed by Barry Boehm, 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.



A typical spiral model

The spiral model is a realistic approach to the development of large-scale systems and software. Because software evolves as the process progresses, the developer and customer better understand and react to risks at each evolutionary level. The spiral model uses prototyping as a risk reduction mechanism but, more important, enables you to apply the prototyping approach at any stage in the evolution of the product. It maintains the systematic stepwise approach suggested by the classic life cycle but incorporates it into an iterative framework that more realistically reflects the real world. The spiral model demands a direct consideration of technical risks at all stages of the project and, if properly applied, should reduce risks before they become problematic.

1.6 Specialized Process Models

Specialized process models is applied when a specialized or narrowly defined software engineering approach is chosen. These model have the characteristics of one or more of the traditional models presented above.

a. Component-Based Development

A component is a modular building block for computer software. 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. The component-based development model leads to software reuse, and reusability provides software engineers with a number of measurable benefits.

Modeling and construction activities begin with the identification of candidate components. These components can be designed as either conventional software modules or object-oriented classes or packages of classes. Regardless of the technology, 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.

b. The Formal Methods Model

The *formal methods model* providing a mathematically based approach to program modeling and the ability to verify that the model is correct. Formal methods eliminate many of the problems such as ambiguity, incompleteness, and inconsistency that are difficult to overcome using other software engineering paradigms. A variation on this approach, called *cleanroom software engineering* is currently applied by some software development organizations.

The Formal Methods Model	Cleanroom software engineering model
▪ Formal methods use set theory and logic notation (i.e. union, intersection, AND, OR, NOT) to create a clear statement of facts (requirements) that can be analyzed to improve (or even prove) correctness and consistency. The bottom line for both methods is the creation of software with extremely low failure rates. ▪ Formal methods translate software requirements into a more formal representation by applying the notation and heuristics of sets to define the data invariant, states, and operations for a system function.	▪ Cleanroom software engineering emphasizes mathematical verification of correctness before program construction commences and certification of software reliability as part of the testing activity. ▪ Cleanroom software engineering uses box structure representation that encapsulates the system (or some aspect of the system) at a specific level of abstraction. Correctness verification is applied once the box structure design is complete. Once correctness has been verified for each box structure, statistical usage testing commences.

Formal methods used in developing computer systems are mathematically based techniques for describing system properties. Such formal methods provide frameworks within which people can specify, develop, and verify systems in a systematic, rather than ad hoc manner.

For example, a block handler of operating system may have various states whose data invariant can be shown in figure:

$$used \cap free = \varnothing \land$$
$$used \cup free = AllBlocks \land$$
$$\forall i: \text{dom } BlockQueue \bullet BlockQueue\ i \subseteq used \land$$
$$\forall i, j: \text{dom } BlockQueue \bullet i \neq j = BlockQueue\ i \cap BlockQueue\ j = \varnothing$$

Cleanroom software engineering makes use of a specialized version of the incremental software model. A "pipeline of software increments" is developed by small independent software teams. As each increment is certified, it is integrated into the whole. Hence, functionality of the system grows with time.

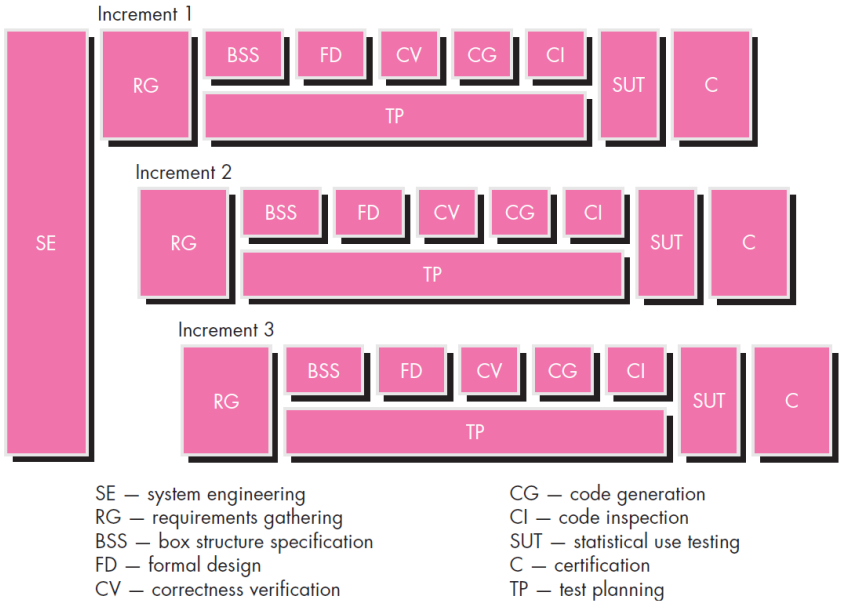


Fig. The cleanroom process model

Some problems on implementation of formal method model and cleanroom software engineering model are:

- The development of formal models 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.

c. Aspect-Oriented Software Development

Aspect-oriented software development (AOSD), often referred to as *aspect-oriented programming* (AOP), is a relatively new software engineering paradigm that provides a process and methodological approach for defining, specifying, designing, and constructing *aspects.* The aspect is customer requirements solely defined for their software system. The aspects may be user interfaces, collaborative work, distribution, persistency, memory management, transaction processing, security, integrity and so on. Components may provide or require one or more "aspect details" relating to a particular aspect, such as a viewing mechanism, extensible affordance and interface kind (user interface aspects); event generation, transport and receiving (distribution aspects); data store/retrieve and indexing (persistency aspects); authentication, encoding and access rights (security aspects); transaction atomicity, concurrency control and logging strategy (transaction aspects); and so on.

1.7 The Unified Process and UML

UML Fundamentals

The Unified Modeling Language (UML) is a standard language for writing software blueprints. The UML may be used to visualize, specify, construct, and document the artifacts of a software-intensive system, as well as for business modeling and other non-software systems. The UML is appropriate for modeling systems ranging from enterprise information systems to distributed Web-based applications and even to hard real time embedded systems. It is a very expressive language, addressing all the views needed to develop and then deploy such systems.

It most directly unifies the methods of Booch, Rumbaugh (OMT), and Jacobson (Objectory), but its reach is wider than that. By 1997, it become a **de facto** standard in the domain of object-oriented analysis and design founded on a wide base of user experience. The UML is called a modeling language, not a method. Most methods consist, at least in principle, of both a modeling language and a process. The *modeling language* is the (mainly graphical) notation that methods use to express designs. The *process* is their advice on what steps to take in doing a design.

- **Unified:** Combines main preceding OO methods (Booch by Grady Booch, OMT by James Rumbaugh and OOSE by Ivar Jacobson).
- **Modeling:** Primarily used for visually modeling systems. Many system views are supported by appropriate model.
- **Language:** It offers syntax through which to express modeled knowledge.

UML is	UML is not
• A language for capturing and expressing knowledge • A tool for system discovery and development • A tool for visual development modeling • A set of well-founded guidelines • A milestone generator • A popular (therefore supported) tool	• A visual programming language or environment • A database specification tool • A development process or SDLC • A panacea or remedy • A quality guarantee

The UML is a language for:

- **Visualizing:** The UML is more than just a bunch of graphical symbols. Rather, behind each symbol in the UML notation is a well-defined semantics. In this manner, one developer can write a model in the UML, and another developer, or even another tool, can interpret that model unambiguously
- **Specifying:** means building models that are precise, unambiguous, and complete.
- **Constructing:** the UML is not a visual programming language, but its models can be directly connected to a variety of programming languages
- **Documenting:** a healthy software organization produces all sorts of artifacts in addition to raw executable code. These artifacts include: *Requirements*, *Architecture*, *Design*, *Source code*, *Project plans*, *Tests*, *Prototypes*, and *Releases*.

UML Diagrams	
Static or Structural Diagram	**Behavioral Diagram**
1. Class Diagram 2. Object Diagram 3. Component Diagram 4. Deployment Diagram	1. Use Case Diagram 2. Interaction Diagram a. Sequential Diagram b. Collaboration Diagram 3. State chart Diagram 4. Activity Diagram

The Unified Process

UML provided the necessary technology to support object-oriented software engineering practice, but it did not provide the process framework to guide project teams in their application of the technology. Over the next few years, Jacobson, Rumbaugh, and Booch developed the *Unified Process*, a framework for object-oriented software engineering using UML. The *phases* of unified process are similar to the generic framework activities as shown in figure. Today, the Unified Process (UP) and UML are widely used on object-oriented projects of all kinds. The iterative, incremental model proposed by the UP can and should be adapted to meet specific project needs.

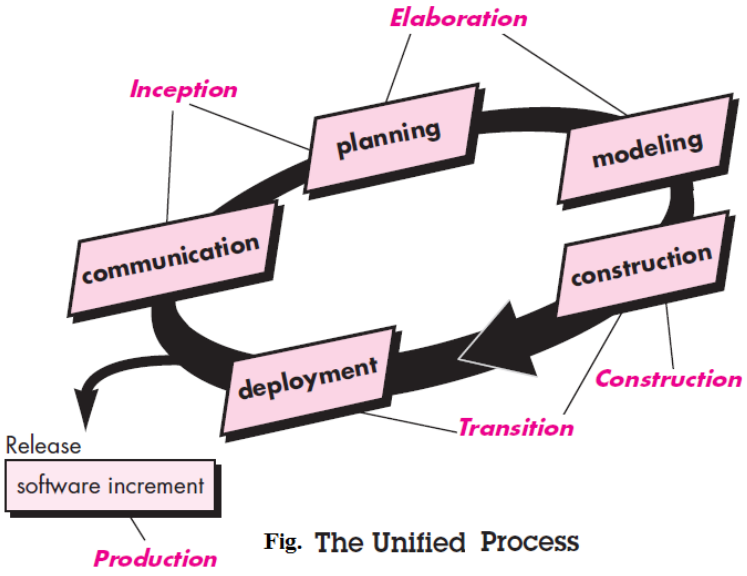


Fig. The Unified Process

1.8 Agile Development

Agile software engineering combines a philosophy and a set of development guidelines to develop the software project as customer need. The philosophy encourages customer satisfaction and early incremental delivery of software; small, highly motivated project teams; informal methods; minimal software engineering work products; and overall development simplicity. The development guidelines stress delivery over analysis and design, and active and continuous communication between developers and customers. The change management is a primary goal of this philosophy. The changes may be from employ turnover, change of customer need, technology change, stakeholder decision change etc.

Software engineers and other project stakeholders (managers, customers, end users) work together on an agile team—a team that is self-organizing and in control of its own destiny. An agile team fosters communication and collaboration among all who serve on it. The basic framework activities—communication, planning, modeling, construction, and deployment—are remains the same for agile development but they transform into a minimal task set that pushes the project team toward construction and delivery.

The conventional wisdom in software development is that the cost of change increases nonlinearly as a project progresses. An agile process reduces the cost of change because software is released in increments and change can be better controlled within an increment. The *increment* indicates the software modules developed by different team in various time interval.

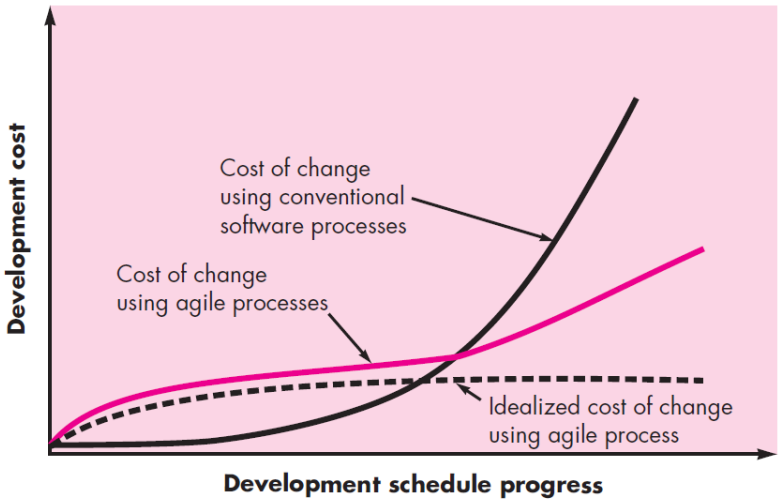


Fig. Change costs as a function of time in development

1. Agile Process

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

a. 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.
b. 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.
c. Analysis, design, construction, and testing are not as predictable (from a planning point of view) as we might like.

These three assumptions indicates the problem of *unpredictability* on software engineering process and hence an agile process must be *adapt* the hidden changes. An agile software process adapt *incrementally* with the help of regular customer feedback. An effective catalyst for customer feedback is an operational prototype or a portion of an operational system. Hence, an *incremental development strategy* should be instituted. *Software increments* (executable prototypes or portions of an operational system) must be delivered in short time periods so that adaptation keeps pace with change (unpredictability). This iterative approach enables the customer to evaluate the software increment regularly, provide necessary feedback to the software team, and influence the process adaptations that are made to accommodate the feedback.

2. Agility Principles

The Agile Alliance defines 12 agility principles for those who want to achieve agility:

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. Agile processes harness change for the customer's competitive advantage.
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.

Not every agile process model applies these 12 principles with equal weight, and some models choose to ignore (or at least downplay) the importance of one or more of the principles. However, the principles define an *agile spirit* that is maintained in each of the process models.

3. Agile Process Models

The history of software engineering taught the lesion that "*The "winners" evolve into best practice, while the "losers" either disappear or merge with the winning models.*" With the introduction of a wide array of agile process models (all follows the principle and paradigms of Agile Software Development), the most common are:

- Extreme Programming (XP),
- Adaptive Software Development (ASD),
- Scrum,
- Dynamic Systems Development Method (DSDM),
- Crystal,
- Feature Drive Development (FDD),
- Lean Software Development (LSD),
- Agile Modeling (AM), and
- Agile Unified Process (AUP).

A. Extreme Programming (XP)

Extreme Programming (XP) is a most widely used approach to agile software development. The ideas and methods associated with XP was written by Kent Beck during the late 1980s. More recently, a variant of XP, called *Industrial XP* (IXP) that refines XP and targets the agile process specifically for use within large organizations.

XP Values

Beck defines a set of five *values* that establish a foundation for all work performed as part of XP—communication, simplicity, feedback, courage, and respect. Each of these values is used as a driver for specific XP activities, actions, and tasks.

- **Communication:** In the XP context, a *metaphor* is "a story that everyone—customers, programmers, and managers— can tell about how the system works". The important concepts and feedback from metaphor help to establish required features and functions for the software.
- **Simplicity:** To achieve *simplicity*, XP restricts developers to design only for immediate needs, rather than consider future needs. The intent is to create a simple design that can be easily implemented in code. If the design must be improved, it can be *refactored* at a later time. Refactoring allows a software engineer to improve the internal structure of a design (or source code) without changing its external functionality or behavior.
- **Feedback:** *Feedback* is derived from three sources: the implemented software itself, the customer, and other software team members. By designing and implementing an effective testing strategy, the software (via test results) provides the agile team with feedback.
- **Courage or Discipline:** There is often significant pressure to design for future requirements because most software teams succumb, arguing that "designing for tomorrow" will save time and effort in the long run. But, an agile XP team must have the discipline (courage) to design for today, recognizing that future requirements may change dramatically, thereby demanding substantial rework of the design and implemented code.
- **Respect:** By following each of these four values, the agile team inculcates *respect* among its members, stakeholders and team members, and indirectly, for the software itself. As they achieve successful delivery of software increments, the team develops growing respect for the XP process.

The XP Process

Extreme Programming uses an object-oriented approach as its preferred development paradigm and encompasses a set of rules and practices that occur within the context of four framework activities: planning, design, coding, and testing. Figure illustrates the XP process and notes some of the key ideas and tasks that are associated with each framework activity.

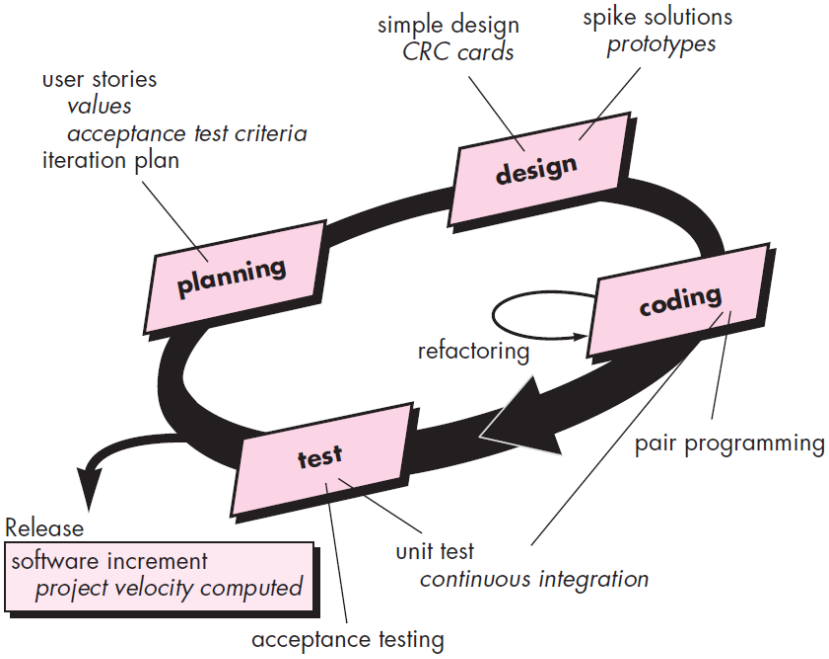


Fig. The Extreme Programming process

B. Adaptive Software Development (ASD)

ASD has been proposed by *Jim Highsmith* as a technique for building complex software and systems. The philosophical underpinnings of ASD focus on human collaboration and team self-organization. ASD "life cycle" that incorporates three phases, speculation, collaboration, and learning.

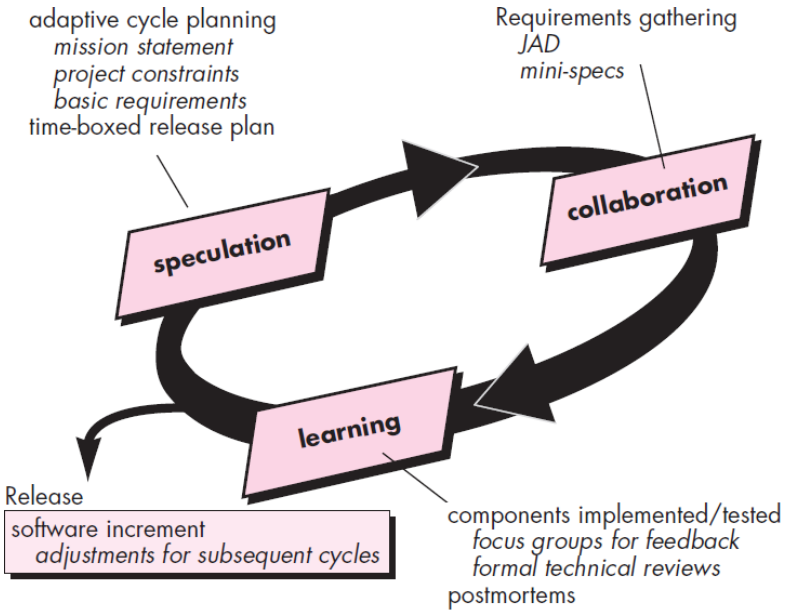


Fig. Adaptive software development

During *speculation*, the project is initiated and *adaptive cycle planning* is conducted. Adaptive cycle planning uses project initiation information—the customer's mission statement, project constraints (e.g., delivery dates or user descriptions), and basic requirements—to define the set of release cycles (software increments) that will be required for the project.

Collaboration encompasses communication and teamwork, but it also emphasizes individualism, because individual creativity plays an important role in collaborative thinking.

Learning will help the agile team to improve their level of real understanding. ASD teams learn in three ways: focus groups, technical reviews, and project postmortems.

C. Scrum

Scrum (the name is derived from an activity that occurs during a rugby match) is an agile software development method that was conceived by *Jeff Sutherland* in the early 1990s. Scrum principles are consistent with the agile manifesto or platform and are used to guide development activities within a process that incorporates the following framework activities: requirements, analysis, design, evolution, and delivery. 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.
- *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 i.e. a period of time allocated to accomplish some task. Changes (e.g., backlog work items) are not introduced during the sprint because 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. A team leader, called a *Scrum master,* leads the meeting and assesses the responses from each person. The Scrum meeting helps the team to uncover potential problems as early as possible. Also, these daily meetings lead to "knowledge socialization" and thereby promote a self-organizing team structure.
- *Demos*—deliver the software increment to the customer so that functionality that has been implemented can be demonstrated and evaluated by the customer.

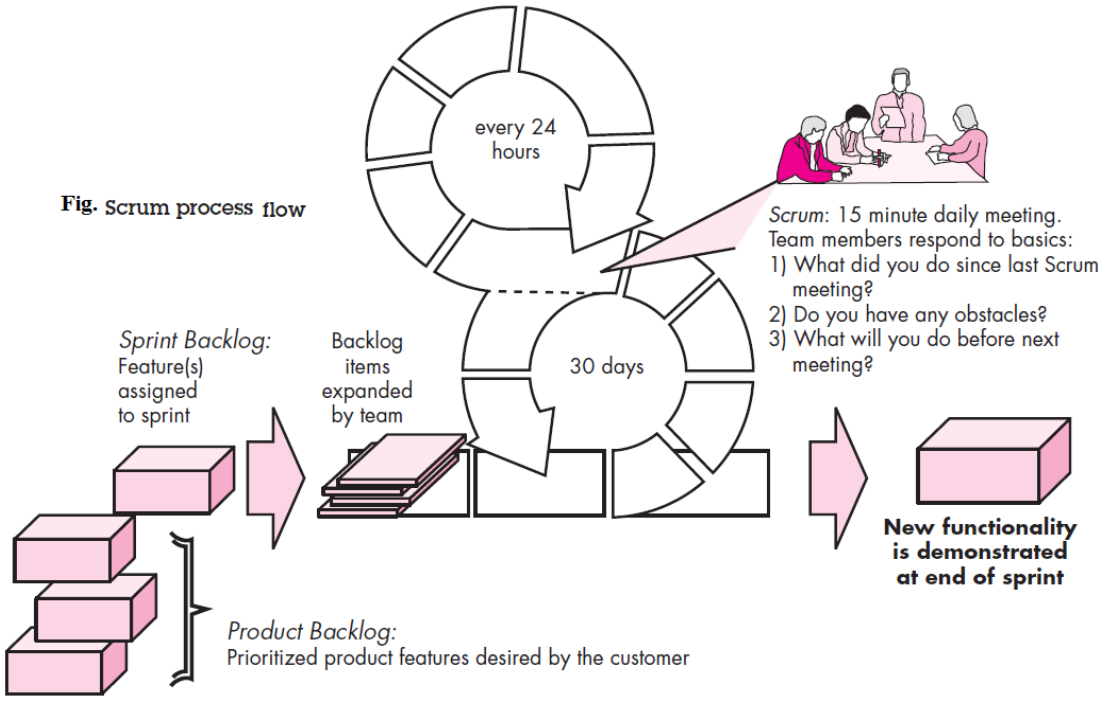


Fig. Scrum process flow

D. Dynamic System Development Method (DSDM)

DSDM provides a framework for building and maintaining systems which meet tight time constraints through the use of incremental prototyping in a controlled project environment. The DSDM philosophy is borrowed from a modified version of the ***Pareto principle***—80 percent of an application can be delivered in 20 percent of the time it would take to deliver the complete (100 percent) application. DSDM also can be combined with XP to provide a combination approach.

The DSDM Consortium (*www.dsdm.org*) is a worldwide group of member companies that collectively take on the role of "keeper" of the method. The consortium has defined an agile process model, called the *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.
- *Functional model iteration*—produces a set of incremental prototypes that demonstrate functionality for the customer. The intent during this iterative cycle is to gather additional requirements by eliciting feedback from users as they exercise the prototype.
- *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. In some cases, *functional model iteration* and *design and build iteration* occur concurrently.
- *Implementation*—provides the operational environment of the design model using any programming language.

E. Feature Driven Development (FDD)

FDD was originally conceived by Peter Coad as a practical process model for object-oriented software engineering and then extended and improved using agile process. This model can applied to moderately sized and larger software projects. FDD adopts a philosophy that (1) emphasizes collaboration among people on an FDD team; (2) manages problem and project complexity using feature-based decomposition followed by the integration of software increments, and (3) communication of technical detail using verbal, graphical, and text-based means. FDD emphasizes software quality assurance activities by encouraging an incremental development strategy, the use of design and code inspections, the application of software quality assurance audits, the collection of metrics, and the use of patterns (for analysis, design, and construction). The FDD approach defines five "collaborating" framework activities as shown in figure.

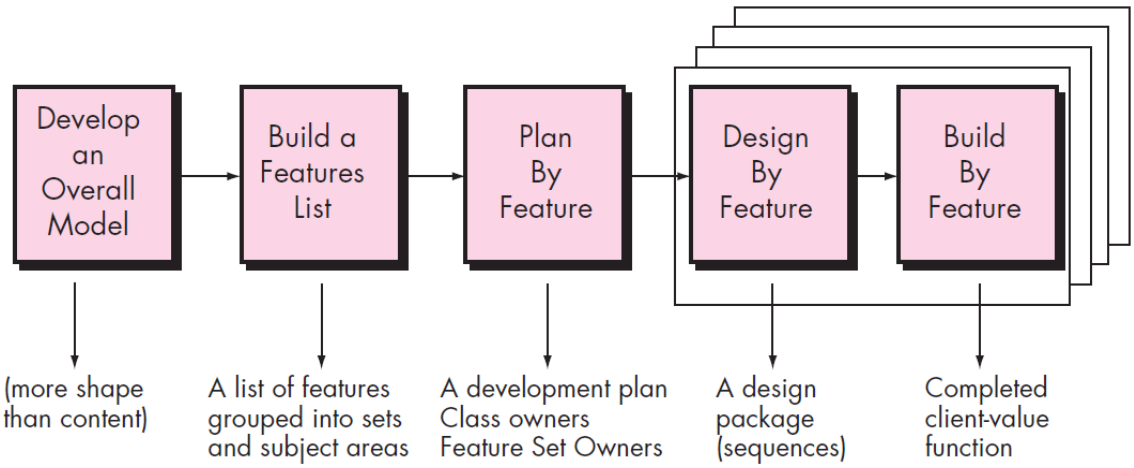


Fig. Feature Driven Development

A *feature* is a client-valued function that can be implemented in two weeks or less. In other word, features are small blocks of deliverable functionality, users can describe them more easily; understand how they relate to one another more readily; and better review them for ambiguity, error, or omissions. To accomplish proper scheduling, FDD defines six milestones during the design and implementation of a *feature*: "design walkthrough, design, design inspection, code, code inspection, promote to build".

Coad and his colleagues suggest the following template for defining a feature:
<action> the **<result>** **<by for of to>** a(n) **<object>,**
where an <object> is "a person, place, or thing.

Examples of features for an e-commerce application might be:

- *Add the product to shopping cart*
- *Display the technical-specifications of the product*
- *Store the shipping-information for the customer.*

F. Lean Software Development (LSD)

Lean Software Development (LSD) has adapted the principles of lean manufacturing to the world of software engineering. The lean principles that inspire the LSD process can be summarized, as *eliminate waste, build quality in, create knowledge, defer commitment, deliver fast, respect people,* and *optimize the whole.*

Each of these principles can be adapted to the software process. For example, *eliminate waste* within the context of an agile software project can be interpreted to mean:

1. Adding no extraneous features or functions,
2. Assessing the cost and schedule impact of any newly requested requirement,
3. Removing any superfluous process steps,
4. Establishing mechanisms to improve the way team members find information,
5. Ensuring the testing finds as many errors as possible,
6. Reducing the time required to request and get a decision that affects the software or the process that is applied to create it, and
7. Streamlining the manner in which information is transmitted to all stakeholders involved in the process.