Internal 1 Long V.IMP QUESTIONS

1)What is the importance of Process Framework? What are the SEA's that are populated in each Framework Activity? (or)
Explain the S/W Engineering process and its activities?

2)Discuss any three Agile Process Models?

3)List the various tasks involved in Requirement Engineering. Explain about each task in short.
(Or)
List and explain the activities related to Requirement Engineering?
(Or)
What are requirements engineering tasks? Explain validating requirements.

4)What are the seven core principles that focus on software engineering practice as a whole ? Briefly explain (or) Explain the core principles of Software engineering practice.

A Process Framework

Diagrammatical representation of software process framework is given in Fig. 2.3.2.

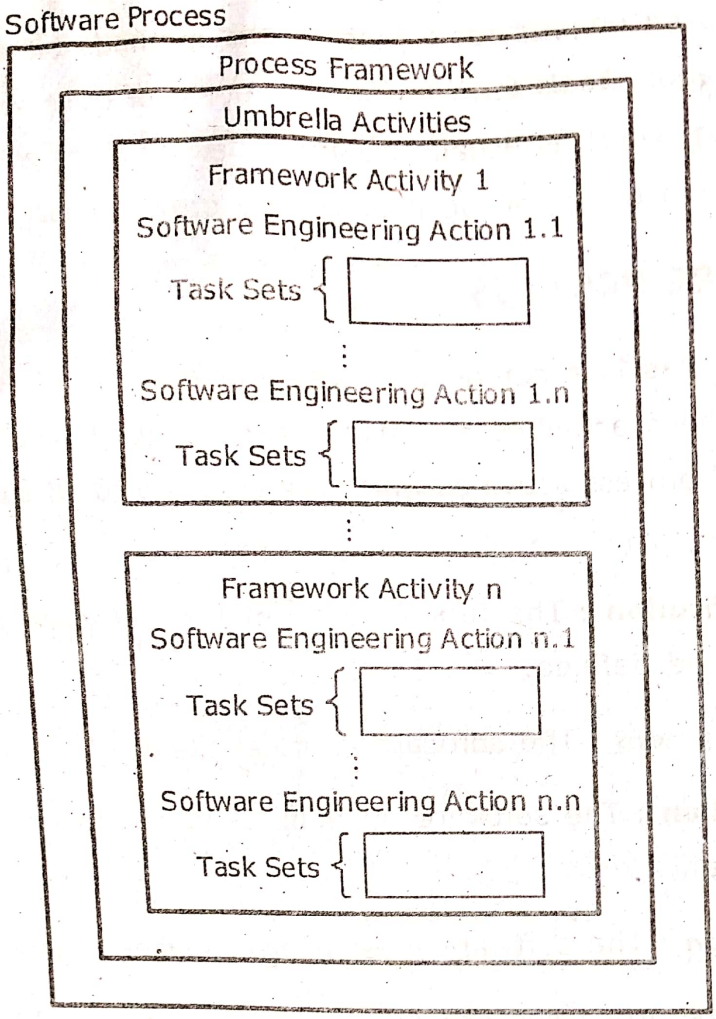

Fig. 2.3.2 A Software Process Framework

A simple diagrammatical overview of a software process framework is shown in Fig. 2.3.2. It plays a major role in every software developmental activity as it forms a base in initiating the developmental process. A software process framework initially estimates certain frame work activities which are applied in every developmental activity, irrespective of size of this developmental activity.

Now consider the Fig. 2.3.2, the process framework includes several umbrella activities, which are useful throughout the software developmental process. Next to umbrella activity is a framework activity which includes definite number of software engineering actions applicable in driving a specific software engineering application. Each of these engineering actions corresponds to the task set which encompasses a list of actions to be applied. Each engineering action can perform a constituent activity which forms essential to the specific software application being developed.

The five generic process framework activities useful in developing several projects is given below,

(1) **Communication :** This refers to a framework activity where, usually the end users (say customers) are communicated and their view related to the project is analyzed. Here, reports related to customer requirement specifications are developed.

(2) **Planning :** In this framework activity, usually the entire work schedule which is going to be implemented in the further is prepared. Hence, various issues addressed during this framework are risk requirement of resources, software schedule, the important products to be developed etc.

(3) **Modeling :** Once we are done with analyzing the customer requirement specifications, then the third framework activity will be modeling. Here, usually UML diagrams are used to represent the project in the form of architecture, so that, it would remain helpful to developers and the customer can gain a sight of the end product.

(4) **Construction :** It is combination of code generation and testing.

(5) **Deployment :** In this frame work activity, usually the developed project is delivered to the end users. They deploy the project and provide relative feedbacks to the developers.

Umbrella Activities of a Software Process Framework

There are eight umbrella activities of a software process framework. These are,

(1) Tracking and controlling software project.

(2) Managing risks.

(3) Software Quality Assurance (SQA).

(4) Formal technical reviews.

(5) Software measurements.

(6) Software Configuration Management (SCM).

(7) Managing reusability factor.

(8) Preparing and producing software work products.

(1) **Tracking and Controlling Software Project :** The first umbrella activity is tracking and control of software project. This activity allows the software team to perform the following tasks,

(i) Assessment of progress of the project against the plan of the project.

(ii) Maintain the schedule of the project by taking appropriate action based on the above assessment.

(2) **Managing Risks :** This activity involves the following tasks,

(i) Evaluation of those risks that can have a serious impact on the final result of the project.

(ii) Assessment of those risks that are likely to effect the product's quality.

(3) **Software Quality Assurance (SQA) :** This activity ensures the software quality by, defining and organizing the activities needed for assurance of quality of software.

(4) **Formal Technical Reviews :** This activity tries to eliminate any errors as quickly as possible so that they don't spread into the other activities. It is done by evaluation of work products of software engineering.

(5) **Software Measurements :** This activity describes as well as gathers measures of process, product and project through which software team can deliver a software according to the needs of the customer.

(6) **Software Configuration Management (SCM) :** SCM is also referred to as Change Management (CM). During software engineering process, SCM defines a set of activities for managing the changes made to the software components. In other words, SCM manages the software components and also provide mechanisms for managing all changes in an efficient and effective manner.

(7) **Managing Reusability Factor :** This activity is incorporated by the software process framework because of the following reasons,

(i) It defines the basic criteria for reusing a work product. Other than this, it also defines a criteria for software components reuse.

(ii) Another reason is, it obtains reusable components by developing the desired methods.

(8) **Prepraing and Producing Software Work Products :** The last and final activity is preparation and production of work product. It includes certain activities that are needed for the creation of the following work products,

(i) Models.

(ii) Logs.

(iii) Documents.

(iv) Lists and forms.

The process models differ in the following ways,

(1) The amount of detail with which the software process is described.

(2) The number of work products that are identified and used in the process.

(3) The amount of work tasks which are defined in every framework activity.

(4) The interdependency factor that exists between several tasks and activities and also in the entire flow of these tasks and activities.

(5) The way in which the activities of the quality assurance are employed.

(6) The way in which the activities of software project tracking and control are employed.

(7) The number of customers and stakeholders that contribute to the project.

(8) The autonomy level provided to every project team.

(9) The number of team organizations and roles that are suggested to be used in the process.

PROCESS ASSESSMENT

Whenever we apply the process patterns to the ongoing software engineering project, it doesn't mean that one software is going to satisfy all the essentials (i.e., on time delivery, customer satisfaction, high quality values etc.,) which should be existed in all the successful softwares. Hence, in order to achieve this, the software patterns should be collaborated with highly valued software engineering practice and the entire process should be sufficiently accessed as required.

Fig. 2.6.1 depicts the software process and methods that are useful during assessment and improvement.

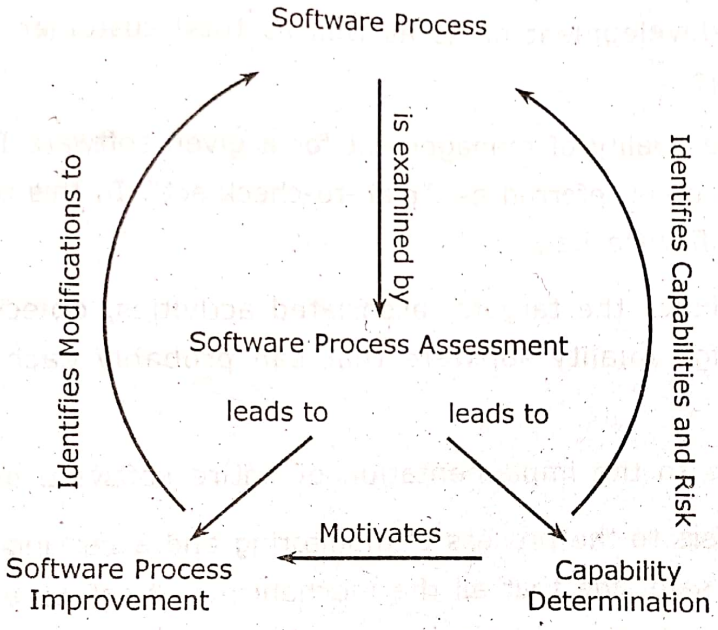

Fig. 2.6.1 Software Process and Methods Applied for Assessment and Improvement

Following are the techniques favouring software process assessment,

(1) **Standard CMMI Assessment Method for Process Improvement (SCAMPI) :** Following are five important steps which form the basis for software process assessment. These are as follows,

(i) Initiating.

(ii) Diagnosing.

(iii) Establishing.

(iv) Acting and

(v) Learning respectively.

In this case, SCAMPI usually relies on SEI CMMI for the requirement of software process assessment.

(2) **CMM-Based Appraisal for Internal Process Improvement (CBA IPI) :** In this case, usually the granularity or maturity of the product or the given organization is assessed considering SEI CMM.

(3) **SPICE (ISO/IEC15504) :** This standard remains effective in deriving software process assessment techniques.

(4) **ISO 9001 : 2000 :** Any software industry, by adapting the ISO 9001 : 2000 standards can reach quality peaks in terms of its products (being manufactured), systems as well as services delivered by it.

This was not initially introduced by International Standard Organization ISO 9001 : 2000 specifies the quality management requirements for a given organization seeking overall development of its as well as total customer satisfaction to be of its primary motive.

To assess the quality of management for a given software ISO 9001 : 2000 has derived a special cycle referred as, "plan-to-check-act". In this phrase each term has got its own significance i.e.,

(i) **Plan :** It includes the targets, associated activities, objectives with an aim to producing high quality software that can probably each complete customer satisfaction.

(ii) **Do :** It refers to the implementation of entire software development process.

(iii) **Check :** It refers to the process of monitoring and accessing the software process to remain at safe side that all the mechanisms of software quality management concepts are thoroughly implemented on the current process of software development.

(iv) **Act :** It refers to the implementation of ideas in favour of improving the current software development process.

WHAT IS AGILITY

Agility is dynamic, content specific, aggressively change embracing and growt oriented.

An agile team is a nimble team able to appropriately respond to changes.

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.

The Agile Alliance defines 12 principles for those who want to achieve agility. They are as follows,

(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 behaviour accordingly.

WHAT IS AN AGILE PROCESS

(1) Agile approaches attempt to overcome the perceived weaknesses of conventional approach.

(2) An agile process is characterized by how it addresses the following,

(i) Changes to the requirements and the customer's priorities.

(ii) Tandem operation of design and construction activities.

(iii) Planning for analysis, design, coding and testing.

(3) Agile processes manage unpredictability by,

(i) "Adapting" themselves to changing environment.

(ii) Adaptations happening in an incremental fashion.

(iii) Increments being delivered in an on-going basis.

(iv) Operating in an iterative manner.

NOTE : Customer feedback plays an important role.

(4) Agile approaches must show "tolerance" for the developer differences. However, there has to be trade-off between discipline and flexibility.

(7) **Self-organization :** In the context of agile development the self-organization having three things. They are,

(i) The agile team organizes itself for the work to be done.

(ii) The agile team organizes the process to best accommodate its local environment.

(iii) The agile team organizes the work schedule to best achieve delivery of the software increment.

The self-organization has a number of technical-benefits, but more importantly it serves to improve collaboration and boost team morale. The team serves as its own management.

AGILE PROCESS MODELS

Agile process models are,

(1) Tend to be more informal.

(2) Emphasize on maneuverability, adaptability and speed.

(3) Appropriate for many types of projects.

Example : Web-based development.

NOTE : Excessively agile processes may be as risky as the excessively rigid processes.

There are a number of different agile process models. They are,

(1) Extreme Programming (XP).

(2) Adaptive Software Development (ASD).

(3) Dynamic Systems Development Method (DSDM).

(4) Scrum Process Model.

(5) Crystal Family.

(6) Feature Driven Development (FDD).

(7) Agile Modeling (AM).

There are many similarities among these approaches. It is important to note all agile models conform to the manifesto for Agile Software Development.

Extreme Programming (XP)

Extreme Programming (XP) is one of the best known agile process. The extreme programming approach was suggested by Kent Beck in 2000. The extreme programming process is explained as follows,

(1) Customer specifies and priorities the system requirements. Customer becomes one of the important members of development team. The developer and customer together prepare a story-card in which customer needs are mentioned.

(2) The developer team then aims to implement the scenarios in the story-card.

(3) After developing the story-card the development team breaks down the total work in small tasks. The efforts and the estimated resources required for these tasks are estimated.

(4) The customer priorities the stories for implementation. If the requirement changes then sometimes unimplemented stories have to be discarded. Then release the complete software in small and frequent releases.

(5) For accommodating new changes, new story-card must be developed.

(6) Evaluate the system along with the customer.

This process is demonstrated by the Fig. 4.4.1.

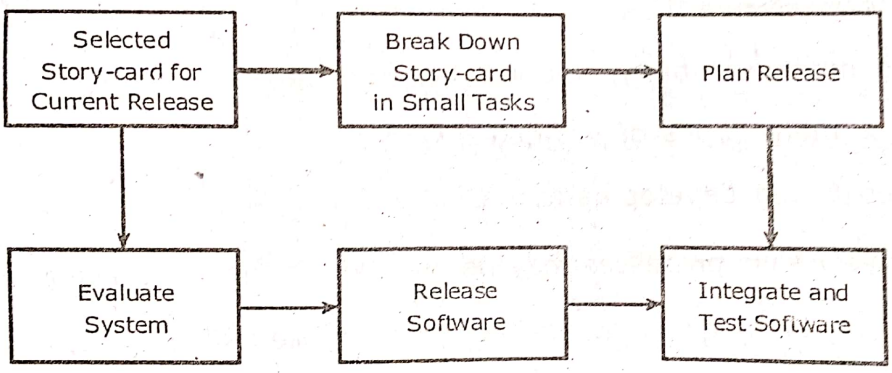

Fig. 4.4.1 Extreme Programming Release Cycle

The Fig. 4.4.2 shows the XP process and notes some of the key ideas and tasks are associated with each framework activity.

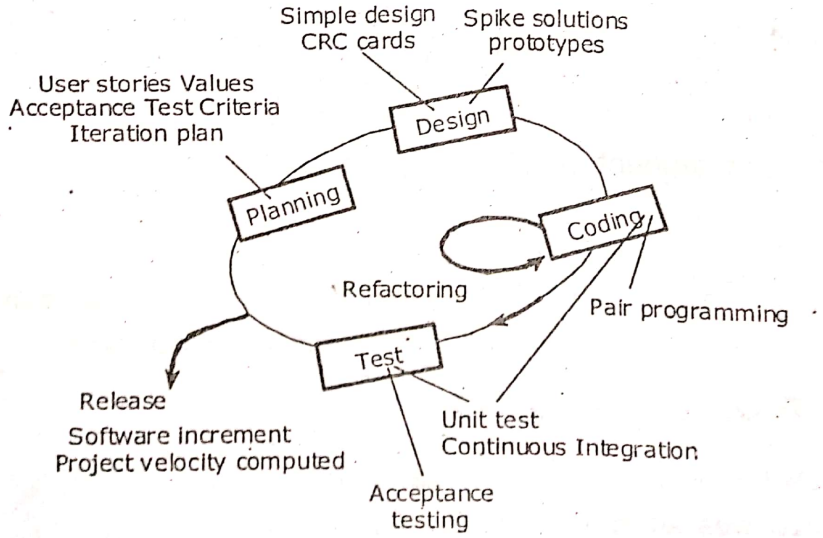

Fig. 4.4.2 Extreme Programming Process

Various rules and practices used in extreme programming are as given in Table 4.4.1.

Table 4.4.1 Rules and Practices used in Extreme Programming

Framework Activities	XP Principle	Description
Planning	User story-cards	Instead of creating a large requirement document user stories are written by the customer in which what they need is mentioned.
	Release planning	A release plan for overall project is prepared from which the iteration plan can be prepared for individual iteration.
	Small releases	The developer breaks down the user stories into small releases and a plan for releasing the small functionalities is prepared.
	Iterative process	Divide the development work into small iterations. Keep the iteration of nearly constant length. Iterative development helps in quick or agile development.
	Stand up meetings	The stand up meetings must be conducted for the current outcomes of the project.
Designing	Simple design	Simple design always takes less time than the complex design. It is always good to keep the things simple to meet the current requirements.
	Spike solution	For answering the tough technical problems create the spike solutions. The goal of these solutions should be to reduce the technical risks.
	Refactoring	Refactoring means reductions in the redundancy, elimination of unused functionalities, redesign the obsolete designs. This will improve the quality of the project.
Coding	Customer availability	The most essential requirement of the XP is availability of the customer. In extreme programming the customer not only helps the developer team but it should be the part of the project.
	Paired Programming	All the code to be included in the project must be coded by groups of two people working at the same computer. This will increase the quality of coding.
	Collective code ownership	By having collective code ownership approach the everyone contributes new ideas and not any single person becomes the bottleneck of the project. Anyone can change any line of code to fix a bug or to refactor.
Testing	Unit testing	The test-first development is done in XP. The test framework that contains the automated test case suite is used to test the code. All the code must be tested using unit testing before its release.
	Continuous integration	As soon as one task is finished integrate it into the whole system. Again after such integration unit testing must be conducted.
	No overtime	Working overtime loses the spirit and motivation of the team. Conduct the release planning meeting to change the project scope or to reschedule the project.

As mentioned earlier in extreme programming instead of creating large requirement documents the user story-card is prepared. For example, if a project is for creating a banking software then the sample story card can be prepared as follows,

(1) Customer makes the balance enquiry.

(2) Customer withdraws some amount.

(3) Customer deposits some amount.

Banking Software Checking Balance, Depositing and Withdrawing

When the customer wants to check the balance then he just enters his account number and the total amount present in his account is displayed to him.

Then customer can withdraw some amount from his account. The limit of amount to be withdrawn must be specified. If the customer crosses that limit then he must be warned. The amount withdrawn must be deducted from the available balance.

The customer can deposit the desired amount in his account. The deposited amount must be added to the available balance.

Note that, the user story is broken into various tasks and simple classes must be created for these tasks. From above example it is clear that the useful class in the project could be 'account'. Various data attributes can be account_no, customer name, address, balance and the useful functions can be deposit( ), withdraw( ), check_balance( ).

Each task can be separately tested using the automated test cases.

Adaptive Software Development (ASD)

Adaptive Software Development (ASD) is a technique for building complex software and systems. This development focus on human collaboration and team self-organization. Self-organization is a property of complex adaptive systems. This arises when individual, independent agents cooperate to create emergent outcomes. An emergent outcome is a property beyond the capability of any individual agent.

Highsmith argues that an agile, adaptive development approach based on collaboration is "as much a source of order in our complex interactions as discipline and engineering".

He defines an ASD life cycle, it contains three phases, speculation, collaboration and learning is as shown in Fig. 4.4.3.

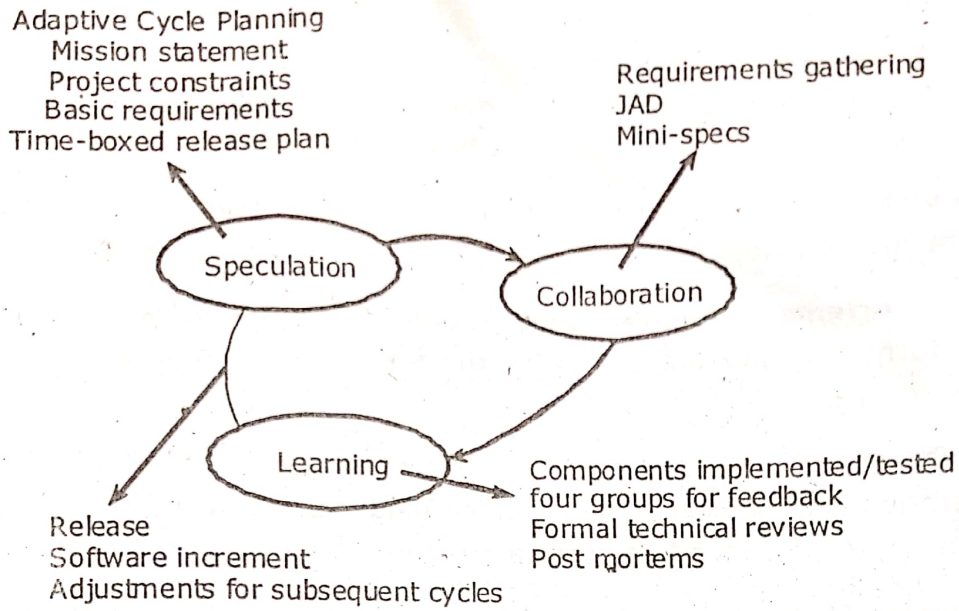

Fig. 4.4.3 Adaptive Software Development

(1) **Speculation :** During this phase, the project is initiated and adaptive cycle planning is conducted. This adaptive cycle planning uses,

(i) Project initiation information.

(ii) The customer's mission statement.

(iii) Project constraints (example, delivery dates or user descriptions).

(iv) Basic requirements are to define the set of release cycles (like software increments) that will be required for the project.

(2) **Collaboration :** In this phase, motivated people work together in a way that multiplies their talent and creative output beyond their absolute numbers. This collaboration approach is a recurring theme in all agile methods. But this phase is not easy. It is not simple communication, although communication is a part of it. It is not only a matter of teamwork, although a jelled team is essential for real collaboration to occur. The people working together must trust one another to,

(i) Criticize without animosity.

(ii) Assist without resentment.

(iii) Work as hard or harder as they do.

(iv) Have the skill set to contribute to the work at hand.

(v) Communicate problems or concerns in a way that leads to effective action.

(3) **Learning :** The members of an ASD team begin to develop the components that are part of an adaptive cycle, the emphasis is on learning as much as it is on progress toward a completed cycle. The learning of an ASD teams in three ways,

(i) **Focus Groups :** The customer and/or end-users provide feedback on software increments that are being delivered. This provides a direct indication of whether or not the product is satisfying business needs.

(ii) **Formal Technical Reviews :** ASD team members review the software components that are developed, improving quality and learning as they proceed.

(iii) **Postmortems :** The ASD team becomes introspective, addressing its own performance and process (with the intent of learning and then improving its approach).

ASD's overall emphasis on the dynamics of self organizing teams, interpersonal collaboration and individual and team learning yield software project teams that have a much higher likelihood of success.

Dynamic Systems Development Method (DSDM)

The Dynamic Systems Development Method (DSDM) is an agile software development approach. This provides a framework for building and maintaining systems which meet tight time constraints through the use of incremental prototyping in a controlled project environment. It is an outgrowth of and extension to Rapid Application Development (RAD) practices. DSDM boasts the best-supported training and documentation of any ASDE, atleast in Europe. DSDM's nine principles include active user involvement, frequent delivery, team decision making, integrated testing throughout the project life cycle and reversible changes in development.

Like extreme programming, adaptive software development, DSDM suggests an iterative software process.

The DSDM consortium 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 mode, called the DSDM life cycle. There are three different interactive cycles, preceded by two additional life cycle activities defined by DSDM life cycle.

(1) ***Feasibility Study :*** This study establisher the basic business requirements and constraints. These are associated with the application to be built and then assesses whether the application is a viable candidate for the DSDM process.

(2) ***Business Study :*** This study establishes the functional and information requirements that will allow the application to provide business value.

(3) ***Functional Model Iteration :*** This study produces a set of incremental prototypes that demonstrate functionality for the customer.

(4) **Design and Build Iteration :** This revisits prototypes built during the functional model iteration to ensure that each has been engineered on a manner that will enable it to provide operational business value for end-users.

(5) **Implementation :** This places the latest software increment into the operational environment. It should be noted that,

(i) The increment may not be 100 percent complete.

(or)

(ii) Changes may be requested as the increment is put into place.

In either case, DSDM development work continues by returning to the function model iteration activity.

DSDM can be combined with extreme programming to provide a combination approach that defined a solid process model i.e., DSDM life cycle with the nuts and bolts practices (XP) that are required to build software reincrements. The ASD concepts of collaboration and self-organizing teams can be adopted to a combined process model.

Scrum Process Model

Scrum is an iterative, incremental process for developing any product or managing any work. It produces a potentially shippable set of functionality at the end of every iteration. It's attributes are,

(1) Scrum is an agile process to manage and control development work.

(2) Scrum is a wrapper for existing engineering practices.

(3) Scrum is a team-based approach to iteratively, incrementally develop systems and products when requirements are rapidly changing.

(4) Scrum is a process that controls the chaos of conflicting interests and needs.

(5) Scrum is a way to improve communications and maximize co-operation.

(6) Scrum is a way to detect and cause the removal of anything that gets in the way of developing and delivering products.

(7) Scrum is a way to maximize productivity.

(8) Scrum is scalable from single projects to entire organizations. Scrum has controlled and organized development and implementation for multiple interrelated products and projects with over a thousand developers and implementers.

(9) Scrum is a way for everyone to feel good about their job, their contributions and that they have done the very best they possibly could.

Fig. 4.4.4 illustrate diagram of scrum flow. Scrum provides daily status on team progress and iterative (every 30 days) reviews of product progress. Everything is visible what's to be worked on, how work is progressing and what has been built supporting management decisions regarding cost, time, quality and functionality. Plus, management is apprised daily what it can do to help the development teams what decisions are needed and what's getting in the way.

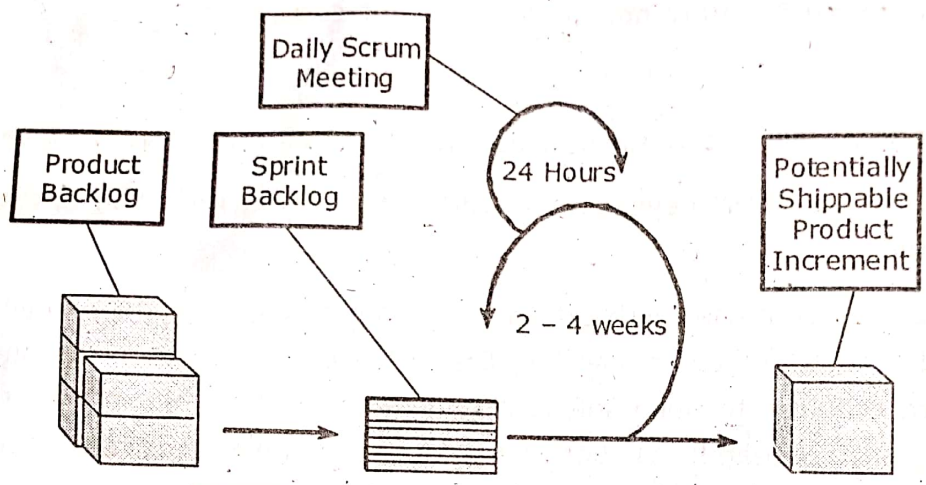

Fig. 4.4.4 Diagram of Scrum Flow

Crystal Family

Alistair Cockburn is the author of the "Crystal" family of people-centered methods. Alistair is a "methodology archaeologist" who has interviewed dozens of project teams worldwide trying to separate what actually works from what people say should work. Alistair and Crystal, focuses on the people aspects of development, collaboration, good citizenship and cooperation. Alistair uses project size, critically and objectives to craft appropriately configured practices for each member of the crystal family of methodologies. The crystal family is actually a set of agile processes that have been proven effective for different types of project. The attention is to allow agile teams to select the member of the crystal family that is most appropriate for their project and environment.

Feature Driven Development (FDD)

(1) FDD emphasizes on defining "features".

(i) A feature is "a client-valued function that can be implemented in two weeks or less".

(2) Since features are "small", the distinguishing aspects of FDD are,

(i) A user can easily,

- Describe them.
- Understand their interrelationships.
- Validate them meaningfully.

(ii) They are easy to review (technically).

(iii) They can be organized by business hierarchy/groups.

(iv) Delivery of increments is every two weeks.

(v) Project planning is driven by feature hierarchy.

(3) Design and construction merge in FDD.

(4) Uses a feature template,

Example, <action> the <result> <by | for | of | to> a(n) <object>

where an <object> is a person, place or thing including roles, moments in time or intervals of time or catalog-entry like descriptions.

The Fig. 4.4.5 shows the FDD approach defines five collaborating framework activities. In FDD these activities are called processes.

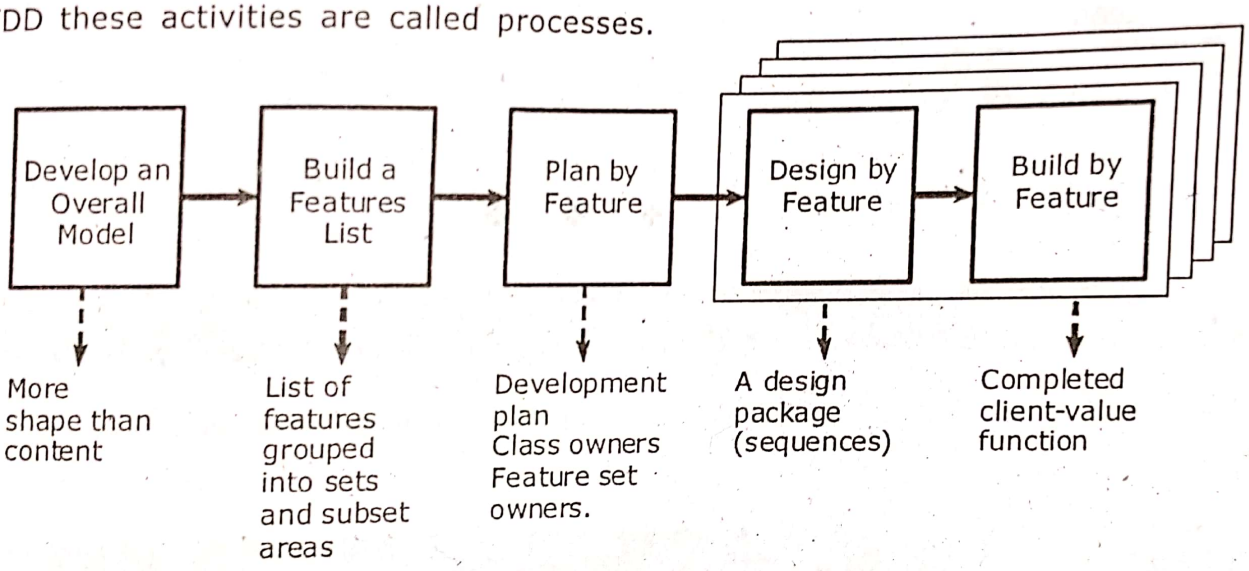

Fig. 4.4.5 Feature Driven Development (FDD) Process

FDD provides greater emphasis on project management guidelines and techniques than many other agile methods.

Agile Modeling (AM)

(1) The scope of large and complex system needs to be modelled so that,

i) All stakeholders can understand "what is to be developed".

ii) Problem can be effectively partitioned.

iii) Quality can be assessed at every step.

(2) "Agile modeling" is a practice based methodology for effective modeling and documentation of software systems.

(3) Agile model is a collection of values, principles and practices that can be used in software development in "light-weight" manner.

(4) Suggested set of agile modeling principles are,

(i) **Model with a Purpose :** A developer who uses agile modeling should have a specific goal in mind before creating the model.

(ii) **Use Multiple Models :** There are many different models and notations that can be used to describe software. Only a small subset is essential for most projects.

(iii) **Travel Light :** As project progresses, keep only relevant models.

(iv) **Content is More Important than Representation :** Modeling should impart information to its intended audience.

(v) **Know the Models and the Tools Well :** Understand the strengths and weaknesses of each model and the tools that are used to create it.

(vi) **Adapt Locally :** The modeling approach should be adapted to the needs of the agile team.

INTRODUCTION TO REQUIREMENTS ENGINEERING

In the old days, requirements engineering used to be known as requirements analysis. Now, requirements engineering is becoming a discipline in its own right (Berry 1998).

A recent study of the Standish group with the American industry confirms that requirements are source of approximately 50% of the problems viz., delays, quality problems or drift of costs.

One of the answers to overcome these problems is requirement engineering. In a traditional engineering approach, a complex system is seen as a sum of optimized systems. Requirement engineering gives a vision of the software product as an optimized system in its environment.

Requirements Engineering (RE) is a relatively recent term that denotes the set of activities that are associated with the elicitation, specification and management of requirements for a computer-based system.

"Requirements Engineering (RE) can be defined as the systematic process of developing requirements through an interactive co-operative process of analyzing the problem, documenting the resulting observations in a variety of representation formats and checking the accuracy of the understanding gained."

A BRIDGE TO DESIGN AND CONSTRUCTION

(1) Requirement engineering helps software engineers to better understand the problem they will work to solve. It includes the set of tasks that lead to an understanding of,

   (i) What will be business impact of the software?

   (ii) What the customer wants exactly?

   (iii) How end user will interact with the system?

(2) Software engineers (sometimes also called as system engineer or analyst) and other project stakeholders (managers, customers and users), all participate in requirement engineering.

(3) Designing and building the elegant computer software will not satisfy customer's requirements. If requirement analysis is wrong, then design will be wrong.

(4) Ultimately coding will be wrong. Finally, software expectations will not match with outcomes. Hence requirement engineering should be carried out carefully.

REQUIREMENTS ENGINEERING TASKS

(1) The requirement engineering is carried out through the execution of seven functions. Some of these requirement engineering functions occur in parallel.

(2) All these functions stress on the customer's needs must be satisfied. All these functions collectively form the strong base for software design and construction. These functions are as follows,

(i) Inception.

(ii) Elicitation.

(iii) Elaboration.

(iv) Negotiation.

(v) Specification.

(vi) Validation.

(vii) Requirements Management.

Inception

(1) Inception means beginning. It is usually said that, 'well beginning is half done'. But it is always problematic for the developer that, 'from where to start'.

(2) The requirement engineering itself is a 'communication intensive' activity.

(3) The customer and developer meet and they decide the overall scope and nature of the problem statement. The aim is,

(i) To have the basic understanding of the problem.

(ii) To know the people who will use the software.

(iii) To know exact nature of problem that is expected from customer's side.

(iv) To maintain effectiveness of preliminary communication.

(v) To have collaboration between customer and developer.

(4) The requirement engineering itself is a 'communication intensive' activity because a requirement gathering is an initial step for design. Hence exact requirements are gathered with customer communication.

The developer must ask following questions,

(1) Who is behind the request for this work?

(2) Who will use the solution?

(3) What will be the economical benefits of a successful solution?

(4) Is there another source of solution for the same problem?

Elicitation

(1) Elicitation means, 'to draw out the truth or reply from anybody'. In relation with requirement engineering, elicitation is a task that helps the customer to define what is required.

(2) To know the objectives of the system or the project to be developed is a critical job. Why it is difficult to get a clear understanding of customer wants? To answer this question, we have a number of problems like,

Problems of Scope : Many times customer states unnecessary technical detail. The unnecessary details may confuse developer instead of giving clarity of overall system objectives.

Problems of Understanding : Sometimes both customer as well as developer has poor understanding of,

(1) What is needed?

(2) Capabilities and limitations of the computing environment.

(3) Understanding of problem domain.

(4) Specify requirements those conflict with other customers and users.

Problems of Volatility : Volatility means 'change from one state to another'. The customer's requirements may change time to time. This is also a major problem while deciding fixed set of requirements.

Elaboration

(1) Elaboration means 'to work out in detail'. The information obtained during inception and elicitation is expanded and modified during elaboration.

(2) Now requirement engineering activity focuses on developing the technical model of the software that will include,

(i) Functions.

(ii) Features.

(iii) Constraints.

(3) Thus, elaboration is a 'analysis modeling' action. This model focuses on 'how the end user will interact with the system'.

Negotiation

Here, 'Negotiation' means 'discussion on financial and other commercial issues'.

In this step customer, user and other stakeholders discuss to decode,

(1) To rank the requirements.

(2) To decide priorities.

(3) To decide risks.

(4) To finalize the project cost.

(5) The impact of above issues on project cost and delivery date.

Specification

(1) The specification is the final work product produced by requirement engineer. The specification may take different forms,

(i) A written document.

(ii) A set of graphical model.

(iii) A formal mathematical model.

(iv) A collection of scenarios.

(v) A prototype.

(vi) Any combination of above.

(2) The specification serves as a foundation of all subsequent software engineering activities.

(3) It totally describes the function and performance of the computer-based system and the constraints those will govern its development.

Validation

All previous work completed will be just useless and meaningless if it is not validated against the customers expectations.

The requirement validation checklist includes,

(1) All requirements are stated clearly?

(2) Are the requirements misinterpreted?

(3) Does the requirements violate any system domain constraints?

(4) Is the system requirement traceable to the system model that is created?

(5) Are the requirements associated with performance, behaviour and operational characteristics?

Requirements Management

(1) A task that helps the project team to identify, control and track requirements and changes to requirements at any time as the project exceeds.

(2) The requirement management begins with identification. Each requirement is assigned a unique identifier.

(3) The traceability table is developed. Each traceability table relates the requirements to one or more aspects of the system or it is environment.

Table 7.3.1 indicates each traceability table relates requirements to one or more aspects of the system or its environment.

Table 7.3.1 Generic Traceability Table

Requirement	Specific Aspect of the Sytem or its Environment								
	A01	A02	A03	A04	A05				Aii
R01			✓		✓				
R02	✓		✓						
R03	✓			✓					✓
R04		✓			✓				
R05	✓	✓		✓					✓
Rnn	✓		✓						

Following are different traceability tables,

(1) **Features Traceability Table :** Shows how requirements relate to important product features observed by customer.

(2) **Source Traceability Table :** Identifies the source of each requirement.

(3) **Dependency Traceability Table :** Shows how requirements are related to one another.

(4) **Subsystem Traceability Table :** Categorizes requirements by the subsystems that they govern.

(5) **Interface Traceability Table :** Shows how requirements relate to internal as well as external interfaces.

Third Set (Questions those Focus on Effectiveness of the Communication Activity)

(1) Are the questions official?

(2) Are the questions related to the problem?

(3) Are the questions 'too many' in quantity?

(4) Can anyone else provide additional information?

(5) Are some questions left to ask?

These questions and others will help to "break the ice" and initiate the communication that is essential to successful elicitation.

ELICITING REQUIREMENTS

Elicitation is a task that helps the customer to define what is required. 'Eliciting requirements' step is carried out by series of following steps,

(1) Collaborative requirements gathering.

(2) Quality function deployment.

(3) User Scenarios.

(4) Elicitation work product.

Collaborative Requirements Gathering

(1) Gathering software requirements is team oriented activity. In this activity, software team members, software engineering manager, members of marketing and product engineering representatives work together. The aim is,

   (i) To identify the problem.

   (ii) To suggest the solution.

   (iii) To negotiate different approaches.

   (iv) To specify the preliminary set of solution requirements.

(2) The meeting for 'collaborative requirements gathering' is conducted to discuss all above issues.

(3) The basic guidelines for conducting a collaborative requirements gathering meeting are,

   (i) Meetings are conducted and attended by both software engineers as well as customers.

   (ii) An agenda is suggested that is formal enough to cover all points these are to be discussed in the meeting.

   (iii) A 'facilitator' may be customer, developer or outsider controls the meeting.

   (iv) A definition mechanism including work sheets, charts, wall stickers, chat rooms, projectors is used.

(4) The goal is,

(i) To identify the problem.

(ii) To suggest the solution.

(iii) To negotiate different approaches.

(iv) To specify the preliminary set of solution requirements in an atmosphere that is conductive to the accomplishment of the goal.

Quality Function Deployment

Quality function deployment is a technique that translates the customer's needs into technical *requirements* for software.

In other words 'Quality function deployment' defines the requirements in a way that maximizes customer satisfaction. Quality function deployment includes three types of requirements. They are as follows,

(1) Normal Requirements : There are the requirements clearly stated by the customer. Hence, these requirements must be present for customer's satisfaction.

For example,

(i) Graphic displays.

(ii) Specific system functions.

(iii) Specified output formats.

(2) Expected Requirements : These requirements are implicit type of requirements. These requirements are not clearly stated by the customer but even then the customer expects them. For example,

(i) The developed system must provide easy human machine interaction.

(ii) The system should be menu driven.

(iii) All hot key buttons help should be provided.

(iv) The system should be 'user friendly'.

(v) The system should be easy to install.

(3) Exciting Requirements : These requirements are neither stated by the customer nor expected. But to make the customer more satisfied, the developer may include some unexpected requirements.

For example, in word processing software development, only standard capabilities are expected. But, it will be a surprise for the customer if 'page layout capabilities' and 'advanced graphical features' are added.

User Scenarios

(1) It is just impossible to move into more technical software engineering activities until the software team understands how these functions and features will be used by different classes and end users.

(2) To understand this, the developer and users create a set of scenarios those will identify all these issues. The scenario is called as 'Use Cases'. Details of Use Case diagrams are included in next chapter.

Elicitation Work Products

(1) The work products produced by requirement elicitation depend upon the size of the system or the system to be built.

(2) The information produced as a consequence of requirements gathering includes,

(i) A statement of need and feasibility.

(ii) A statement of scope for the system or product.

(iii) A list of customers, users and other stakeholders who participated in requirement elicitation.

(iv) Description of system's technical environment.

(v) The list of requirements and domain constraints.

(vi) The set of scenarios.

(vii) Any prototype developed to define requirements clearly.

VALIDATING REQUIREMENTS

The analysis model is created for each element and it is examined for consistency, omissions and ambiguity.

The requirements represented by the model are prioritized by the customer and grouped within requirements packages that will be implemented as software increments and delivered to the customer.

A review of the analysis model addresses the following questions,

(1) Is each requirement consistent with the overall objective for the system/product?

(2) Have all requirements been specified at the proper level of abstraction? That is, do some requirements provide a level of technical detail that is inappropriate at this stage?

(3) Is the requirement really necessary or does it represent an add-on feature that may not be essential to the objective of the system?

(4) Is each requirement bounded and unambiguous?

(5) Does each requirement have attribution? That is, is a source (generally, a specific individual) noted for each requirement?

(6) Do any requirements conflict with other requirements?

(7) Is each requirement achievable in the technical environment that will house the system or product?

(8) Is each requirement testable, once implemented?

(9) Does the requirements model properly reflect the information, function nd behaviour of the system to be built?

(10) Has the requirements model been "partitioned" in a way that exposes progressively more detailed information about the system?

(11) Have requirements patterns been used to simplify the requirements model? Have all patterns been properly validated? Are all patterns consistent with customer requirements.

These and other questions should be asked and answered to ensure that the requirements model is an accurate reflection of the customer's needs and that it provides a solid foundation for design.

Core Principles

The dictionary defines the word *principle* as "an important underlying law or assumption required in a system of thought." Throughout this book we discuss principles at many different levels of abstraction. Some focus on software engineering as a whole, others consider a specific generic framework activity (e.g., customer communication), and still others focus on software engineering actions (e.g., architectural design) or technical tasks (e.g., write a usage scenario). Regardless of their level of focus, principles help us establish a mind set for solid software engineering practice. They are important for that reason.

David Hooker [HOO96] has proposed seven core principles that focus on software engineering practice as a whole. They are reproduced below:[3]

The First Principle: *The Reason It All Exists*

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

The Second Principle: *KISS (Keep It Simple, Stupid!)*

Software design is not a haphazard process. There are many factors to consider in any design effort. *All design should be as simple as possible, but no simpler.* This facilitates having a more easily understood, and easily maintained system. This is not to say that features, even internal features, should be discarded in the name of simplicity. Indeed, the more elegant designs are usually the simple ones. Simple also does not mean "quick and dirty." In fact, it often takes a lot of thought and work over multiple iterations to simplify. The pay-off is software that is more maintainable and less error-prone.

> "There is a certain majesty in simplicity which is far above all the quaintness of wit."
>
> Alexander Pope (1688–1744)

The Third Principle: *Maintain the Vision*

A clear vision is essential to the success of a software project. Without one, a project almost unfailingly ends up being "of two [or more] minds" about itself. Without conceptual integrity, a system threatens to become a patchwork of incompatible designs, held together by the wrong kind of screws. . . .

Compromising the architectural vision of a software system weakens and will eventually break even a well-designed system. Having an empowered architect

who can hold the vision and enforce compliance helps ensure a very successful software project.

The Fourth Principle: *What You Produce, Others Will Consume*

Seldom is an industrial-strength software system constructed and used in a vacuum. In some way or other, someone else will use, maintain, document, or otherwise depend on being able to understand your system. So, *always specify, design, and implement knowing someone else will have to understand what you are doing.* The audience for any product of software development is potentially large. Specify with an eye to the users. Design, keeping the implementers in mind. Code with concern for those who must maintain and extend the system. Someone may have to debug the code you write, and that makes them a user of your code. Making their job easier adds value to the system.

The Fifth Principle: *Be Open to the Future*

A system with a long lifetime has more value. In today's computing environments, where specifications change on a moment's notice and hardware platforms are obsolete after just a few months, software lifetimes are typically measured in months instead of years. However, true "industrial-strength" software systems must endure far longer. To do this successfully, these systems must be ready to adapt to these and other changes. Systems that do this successfully are those that have been designed this way from the start. *Never design yourself into a corner.* Always ask "what if," and prepare for all possible answers by creating systems that solve the general problem, not just the specific one.[4] This could very possibly lead to the reuse of an entire system.

The Sixth Principle: *Plan Ahead for Reuse*

Reuse saves time and effort.[5] Achieving a high level of reuse is arguably the hardest goal to accomplish in developing a software system. The reuse of code and designs has been proclaimed as a major benefit of using object-oriented technologies. However, the return on this investment is not automatic. To leverage the reuse possibilities that object-oriented [or conventional] programming provides requires forethought and planning. There are many techniques to realize reuse at every level of the system development process. Those at the detailed design and code level are well known and documented. New literature is addressing the reuse of design in the form of software patterns. However, this is just part of the battle.

Communicating opportunities for reuse to others in the organization is paramount. How can you reuse something that you don't know exists? *Planning ahead for reuse reduces the cost and increases the value of both the reusable components and the systems into which they are incorporated.*

The Seventh Principle: *Think!*

This last Principle is probably the most overlooked. *Placing clear, complete thought before action almost always produces better results.* When you think about something, you are more likely to do it right. You also gain knowledge about how to do it right again. If you do think about something and still do it wrong, it becomes valuable experience. A side effect of thinking is learning to recognize when you don't know something, at which point you can research the answer. When clear thought has gone into a system, value comes out. Applying the first six Principles requires intense thought, for which the potential rewards are enormous.

If every software engineer and every software team simply followed Hooker's seven principles, many of the difficulties we experience in building complex computer-based systems would be eliminated.