# UNIT-2

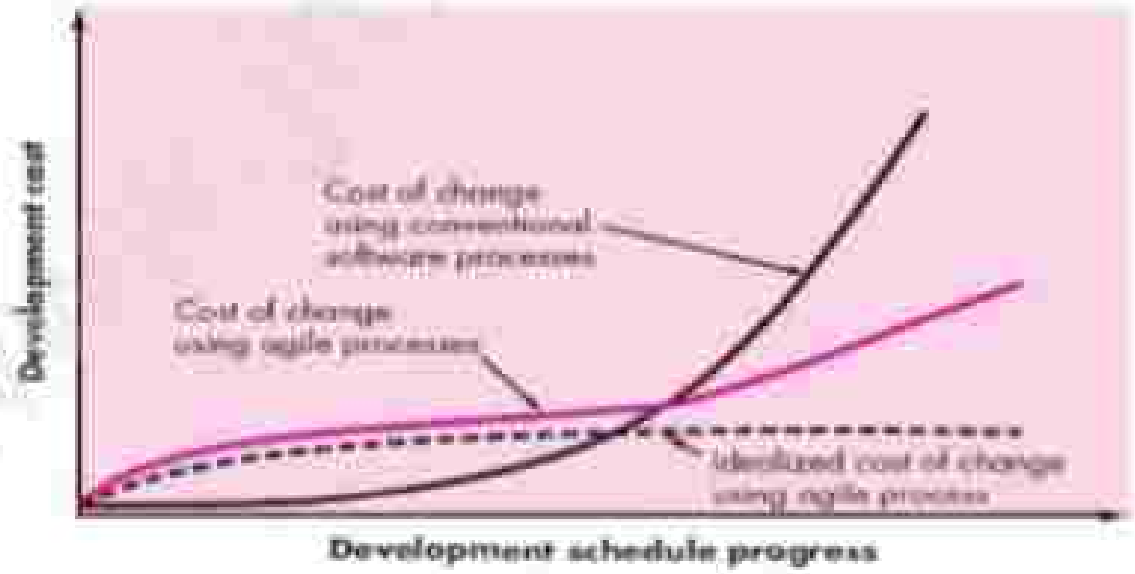
## AGILE DEVELOPMENT

### WHAT IS AGILITY?

Agile is a software development methodology to build software incrementally using short iterations of 1 to 4 weeks so that the development process is aligned with the changing business needs. An agile team is a nimble team able to appropriately respond to changes. Change is what software development is very much about. Changes in the software being built, changes to the team members, changes because of new technology, changes of all kinds that may have an impact on the product they build or the project that creates the product. Support for changes should be built-in everything we do in software, something we embrace because it is the heart and soul of software. An agile team recognizes that software is developed by individuals working in teams and that the skills of these people, their ability to collaborate is at the core for the success of the project.

### AGILITY AND THE COST OF CHANGE

An agile process reduces the cost of change because software is released in increments and change can be better controlled within an increment. Agility argue that a well-designed agile process "flattens" the cost of change curve shown in following figure, allowing a software team to accommodate changes late in a software project without dramatic cost and time impact. When incremental delivery is coupled with other agile practices such as continuous unit testing and pair programming, the cost of making a change is attenuated. Although debate about the degree to which the cost curve flattens is ongoing, there is evidence to suggest that a significant reduction in the cost of change can be achieved.



## AGILE PROCESS

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

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

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

3. Analysis, design, construction, and testing are not as predictable.

## Agility Principles.

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.

**Human Factors**

Agile development focuses on the talents and skills of individuals, molding the process to specific people and teams." The key point in this statement is that the process molds to the needs of the people and team.

- **Competence.** In an agile development context, "competence" encompasses innate talent, specific software-related skills, and overall knowledge of the process that the team has chosen to apply. Skill and knowledge of process can and should be taught to all people who serve as agile team members.
- **Common focus.** Although members of the agile team may perform different tasks and bring different skills to the project, all should be focused on one goal—to deliver a working software increment to the customer within the time promised. To achieve this goal, the team will also focus on continual adaptations (small and large) that will make the process fit the needs of the team.
- **Collaboration.** Software engineering (regardless of process) is about assessing, analyzing, and using information that is communicated to the software team; creating information that will help all stakeholders understand the work of the team; and building information (computer software and relevant databases) that provides business value for the customer. To accomplish these tasks, team members must collaborate—with one another and all other stakeholders.
- **Decision-making ability.** Any good software team (including agile teams) must be allowed the freedom to control its own destiny. This implies that the team is given autonomy—decision-making authority for both technical and project issues.
- **Fuzzy problem-solving ability.** Software managers must recognize that the agile team will continually have to deal with ambiguity and will continually be buffeted by change.
- **Mutual trust and respect.** The agile team must become what DeMarco and Lister call a "jelled" team. A jelled team exhibits the trust and respect that are necessary to make them "so strongly knit that the whole is greater than the sum of the parts."
- **Self-organization.** In the context of agile development, self-organization implies three things: (1) the agile team organizes itself for the work to be done, (2) the team organizes the process to best accommodate its local environment, (3) the team organizes the work schedule to best achieve delivery of the software increment. Self-organization has a number of technical benefits, but more importantly, it serves to improve collaboration and boost team morale.

## EXTREME PROGRAMMING (XP)

**Extreme Programming (XP)** the most widely used approach to agile software development, emphasizes business results first and takes an incremental, get-something-started approach to building the product, using continual testing and revision.

**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**.

In order to achieve effective **communication** between software engineers and other stakeholders, XP emphasizes close, yet informal collaboration between customers and developers, the establishment of effective metaphors3 for communicating important concepts, continuous feedback, and the avoidance of voluminous documentation as a communication medium.

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 re factored at a later time.

**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 provides the agile team with feedback. XP makes use of the unit test as its primary testing tactic. As each class is developed, the team develops a unit test to exercise each operation according to its specified functionality.

Beck argues that strict adherence to certain XP practices demands courage. A better word might be **discipline**. 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.

By following each of these values, the agile team inculcates **respect** among it members, between other 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. Following figure illustrates the XP process and notes some of the key ideas and tasks that are associated with each framework activity.

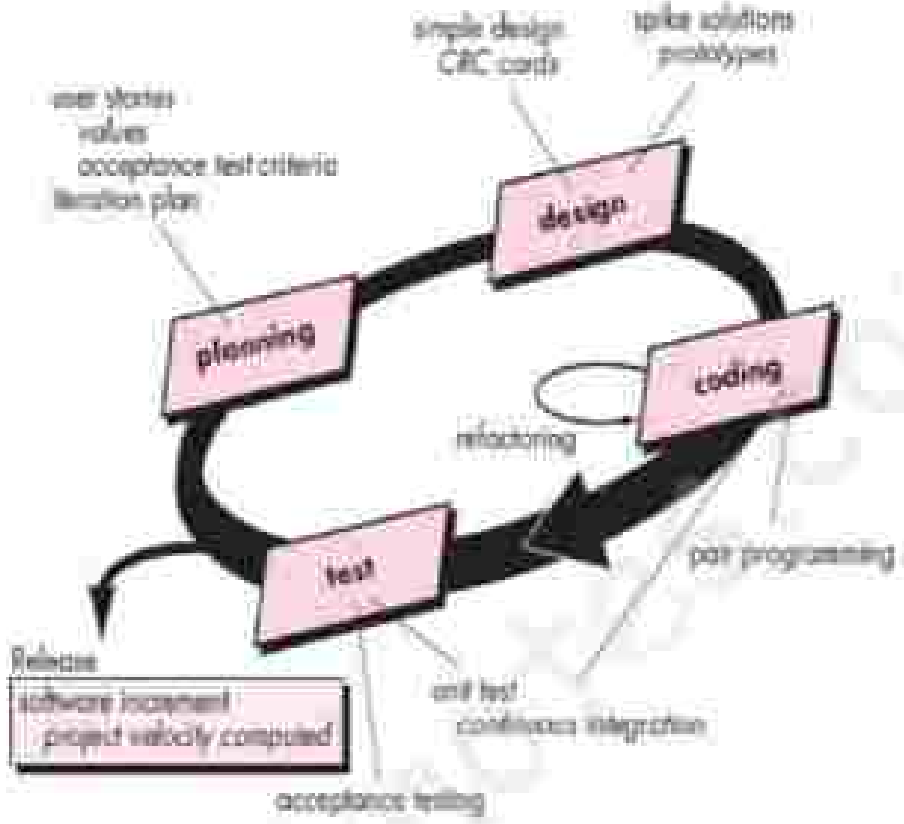



**Fig : The Extreme Programming process**

**Key XP activities are**

- **Planning.** The planning activity (also called the planning game) begins with listening- a requirements gathering activity that enables the technical members of the XP team to understand the business context for the software and to get a broad feel for required output and major features and functionality.
- **Design.** XP design rigorously follows the KIS (keep it simple) principle. A simple design is always preferred over a more complex representation. In addition, the design provides implementation guidance for a story as it is written—nothing less, nothing more. The design of extra functionality If a difficult design problem is encountered as part of the design of a story, XP recommends the immediate creation of an operational prototype of that portion of the design. Called a **spike solution**, the design prototype is implemented and evaluated. XP encourages **refactoring**—a construction technique that is also a method for design optimization.
  Fowler describes refactoring in the following manner: Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of

the code yet improves the internal structure. It is a disciplined way to clean up code [that minimizes the chances of introducing bugs].

- **Coding**. After stories are developed and preliminary design work is done, the team does not move to code, but rather develops a series of unit tests that will exercise each of the stories that is to be included in the current release. Once the code is complete, it can be unit-tested immediately, thereby providing instantaneous feedback to the developers.

A key concept during the coding activity is pair programming. XP recommends that two people work together at one computer workstation to create code for a story. This provides a mechanism for real time problem solving (two heads are often better than one) and real-time quality assurance.

- **Testing.** The creation of unit tests before coding commences is a key element of the XP approach. The unit tests that are created should be implemented using a framework that enables them to be automated. This encourages a regression testing strategy whenever code is modified. Wells states: "Fixing small problems every few hours takes less time than fixing huge problems just before the deadline."

**XP acceptance tests**, also called customer tests, are specified by the customer and focus on overall system features and functionality that are visible and reviewable by the customer. Acceptance tests are derived from user stories that have been implemented as part of a software release.

**Industrial XP**

Joshua Kerievsky describes Industrial Extreme Programming (IXP) in the following manner: "IXP is an organic evolution of XP. It is imbued with XP's minimalist, customer-centric, test-driven spirit. IXP differs most from the original XP in its greater inclusion of management, its expanded role for customers, and its upgraded technical practices." IXP incorporates six new practices that are designed to help ensure that an XP project works successfully for significant projects within a large organization.

- **Readiness assessment.** Prior to the initiation of an IXP project, the organization should conduct a readiness assessment. The assessment ascertains whether (1) an appropriate development environment exists to support IXP, (2) the team will be populated by the proper set of stakeholders, (3) the organization has a distinct quality program and supports continuous improvement, (4) the organizational culture will support the new values of an agile team, and (5) the broader project community will be populated appropriately.
- **Project community**. Classic XP suggests that the right people be used to populate the agile team to ensure success. The implication is that people on the team must be well-trained, adaptable and skilled, and have the proper temperament to contribute to a self-

organizing team. When XP is to be applied for a significant project in a large organization, the concept of the "team" should morph into that of a community. A community may have a technologist and customers who are central to the success of a project as well as many other stakeholders (e.g., legal staff, quality auditors, manufacturing or sales types) who "are often at the periphery of an IXP project yet they may play important roles on the project". In IXP, the community members and their roles should be explicitly defined and mechanisms for communication and coordination between community members should be established.

- **Project chartering.** The IXP team assesses the project itself to determine whether an appropriate business justification for the project exists and whether the project will further the overall goals and objectives of the organization. Chartering also examines the context of the project to determine how it complements, extends, or replaces existing systems or processes.
- **Test-driven management.** An IXP project requires measurable criteria for assessing the state of the project and the progress that has been made to date. Test-driven management establishes a series of measurable "destinations" and then defines mechanisms for determining whether or not these destinations have been reached.
- **Retrospectives.** An IXP team conducts a specialized technical review after a software increment is delivered. Called a retrospective, the review examines "issues, events, and lessons-learned" across a software increment and/or the entire software release. The intent is to improve the IXP process.
- **Continuous learning.** Because learning is a vital part of continuous process improvement, members of the XP team are encouraged (and possibly, incented) to learn new methods and techniques that can lead to a higher quality product.

## OTHER AGILE PROCESS MODELS

Other agile process models have been proposed and are in use across the industry. Among the most common are:

- Adaptive Software Development (ASD)
- Scrum
- Dynamic Systems Development Method (DSDM)
- Crystal
- Feature Drive Development (FDD)
- Lean Software Development (LSD)
- Agile Modeling (AM)

### Adaptive Software Development (ASD)

Adaptive Software Development (ASD) Adaptive Software Development (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. **High smith** 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" that incorporates three phases, speculation, collaboration, and learning.

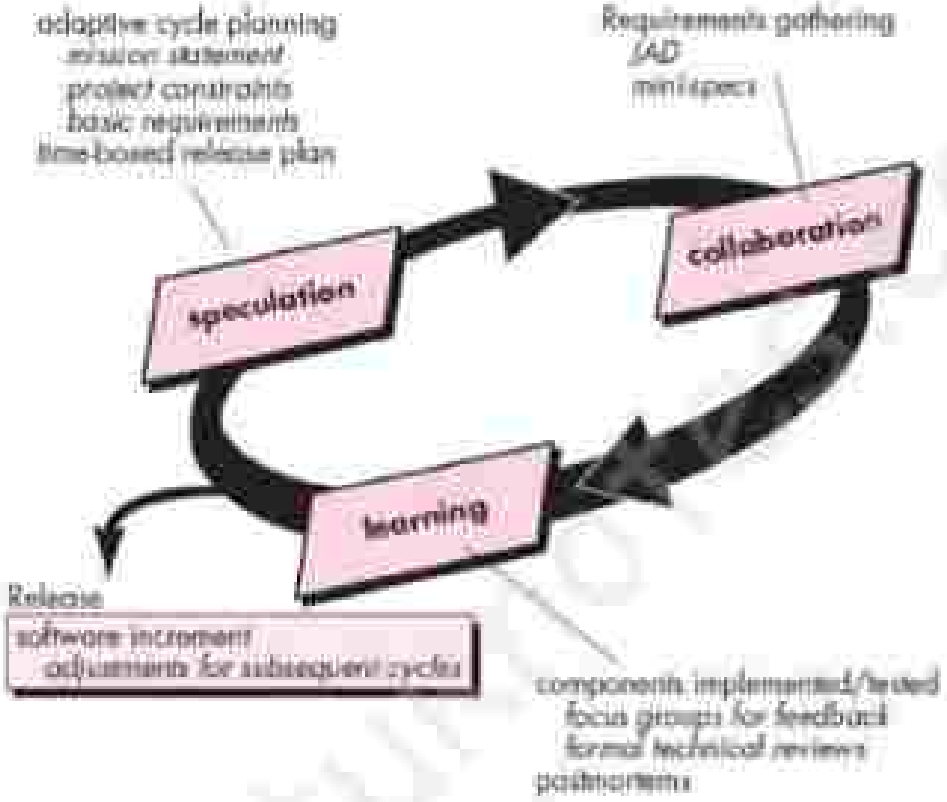



**Fig : Adaptive software development**

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.

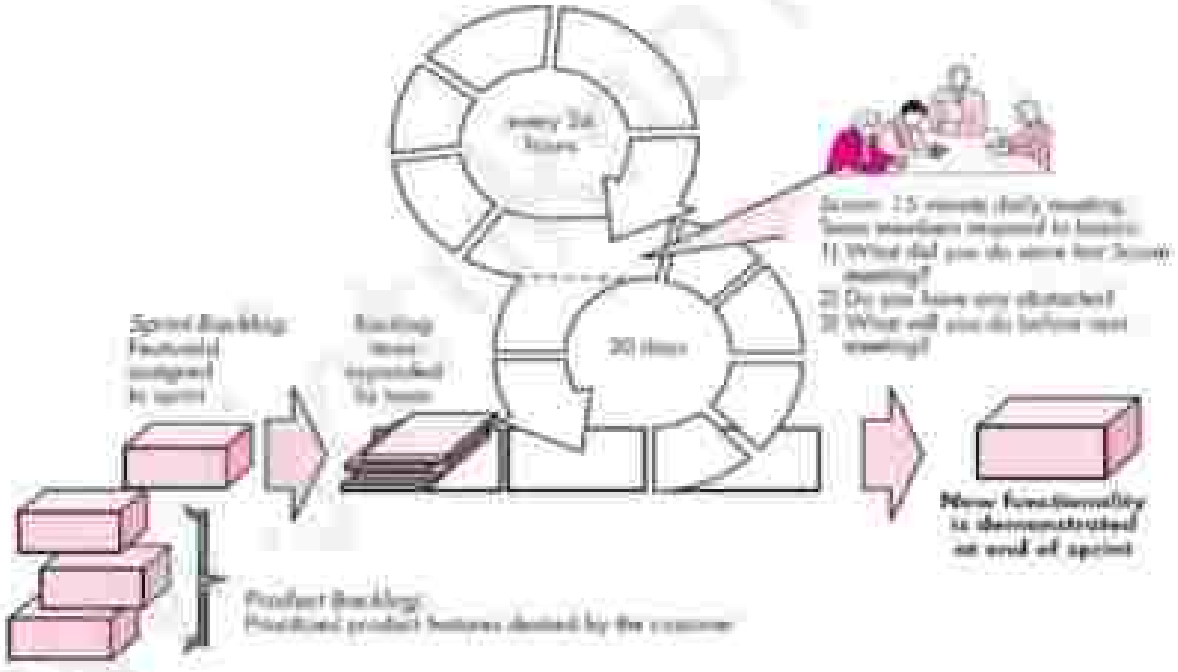
Motivated people use **collaboration** in a way that multiplies their talent and creative output beyond their absolute numbers. This approach is a recurring theme in all agile methods. But collaboration is not easy. It encompasses communication and teamwork, but it also emphasizes individualism, because individual creativity plays an important role in collaborative thinking. It is, above all, a matter of trust. People working together must trust one another to (1) criticize without animosity, (2) assist without resentment, (3) work as hard as or harder than they do, (4) have the skill set to contribute to the work at hand, and (5) communicate problems or concerns in a way that leads to effective action.

As 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.

ASD teams **learn** in three ways: focus groups, technical reviews , and project postmortems. 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.

**Scrum**

Scrum is an agile software development method that was conceived by Jeff Sutherland and his development team in the early 1990s. Scrum principles are consistent with the agile manifesto and are used to guide development activities within a process that incorporates the following framework activities: requirements, analysis, design, evolution, and delivery. Within each framework activity, work tasks occur within a process pattern called a sprint. The work conducted within a sprint is adapted to the problem at hand and is defined and often modified in real time by the Scrum team. The overall flow of the Scrum process is illustrated in following figure




**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. The product manager assesses the backlog and updates priorities as required.

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

• **Scrum meetings**—are short (typically 15 minutes) meetings held daily by the Scrum team. Three key questions are asked and answered by all team members o What did you do since the last team meeting? o What obstacles are you encountering? o What do you plan to accomplish by the next team meeting? 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"

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

**Dynamic Systems Development Method (DSDM)**

The Dynamic Systems Development Method (DSDM) is an agile software development approach that "provides a framework for building and maintaining systems which meet tight time constraints through the use of incremental prototyping in a controlled project environment" 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 is an iterative software process in which each iteration follows the 80 percent rule. 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; also, defines the basic application architecture and identifies the maintainability requirements for the application.
- **Functional model iteration**—produces a set of incremental prototypes that demonstrate functionality for the customer
- **Design and build iteration**—revisits prototypes built during functional model iteration to ensure that each has been engineered in a manner that will enable it to provide

operational business value for end users. In some cases, functional model iteration and design and build iteration occur concurrently.

- **Implementation**—places the latest software increment into the operational environment. It should be noted that (1) the increment may not be 100 percent complete or (2) changes may be requested as the increment is put into place. In either case, DSDM development work continues by returning to the functional model iteration activity.

**Crystal**

Alistair Cockburn and Jim Highsmith created the Crystal family of agile methods in order to achieve a software development approach that puts a premium on "maneuverability" during what Cockburn characterizes as "a resource limited, cooperative game of invention and communication, with a primary goal of delivering useful, working software and a secondary goal of setting up for the next game" .

The Crystal family is actually a set of example agile processes that have been proven effective for different types of projects. The intent 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)**

Feature Driven Development (FDD) was originally conceived by Peter Coad and his colleagues as a practical process model for object-oriented software engineering. Stephen Palmer and John Felsing have extended and improvedCoad's work, describing an adaptive, agile process that can be applied to moderately sized and larger software projects.

Like other agile approaches, 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.

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

• Because 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.

• Features can be organized into a hierarchical business-related grouping.

• Since a feature is the FDD deliverable software increment, the team develops operational features every two weeks.

• Because features are small, their design and code representations are easier to inspect effectively.

• Project planning, scheduling, and tracking are driven by the feature hierarchy, rather than an arbitrarily adopted software engineering task set.

Coad and his colleagues suggest the following template for defining a feature:

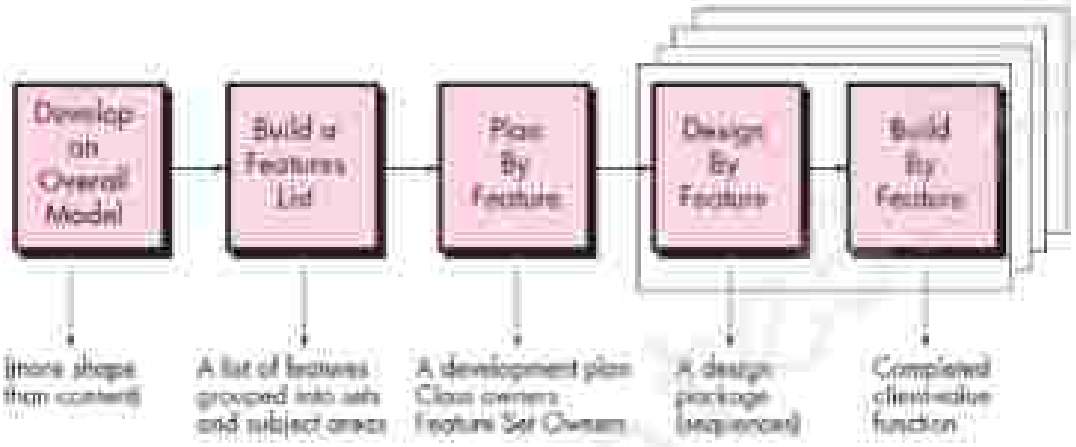



**Fig : Feature Driven Development (FDD)**

FDD provides greater emphasis on project management guidelines and techniques than many other agile methods. FDD defines six milestones during the design and implementation of a feature: **"design walkthrough, design, design inspection, code, code inspection, promote to build"** .

## 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.

## Agile Modeling (AM)

Agile Modeling (AM) is a practice-based methodology for effective modeling and documentation of software-based systems. Simply put, Agile Modeling (AM) is a collection of values, principles, and practices for modeling software that can be applied on a software

development project in an effective and light-weight manner. Agile models are more effective than traditional models because they are just barely good, they don't have to be perfect.Agile Modeling suggests a wide array of "core" and "supplementary" modeling principles, those that make AM unique are :

• **Model with a purpose**. A developer who uses AM should have a specific goal in mind before creating the model. Once the goal for the model is identified, the type of notation to be used and level of detail required will be more obvious.

• **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. AM suggests that to provide needed insight, each model should present a different aspect of the system and only those models that provide value to their intended audience should be used.

• **Travel light**. As software engineering work proceeds, keep only those models that will provide long-term value and jettison the rest. Every work product that is kept must be maintained as changes occur. This represents work that slows the team down. Ambler notes that "Every time you decide to keep a model you trade-off agility for the convenience of having that information available to your team in an abstract manner.

• **Content is more important than representation**. Modeling should impart information to its intended audience. A syntactically perfect model that imparts little useful content is not as valuable as a model with flawed notation that nevertheless provides valuable content for its audience.

• **Know the models and the tools you use to create them**. Understand the strengths and weaknesses of each model and the tools that are used to create it.

• **Adapt locally**. The modeling approach should be adapted to the needs of the agile team.

**Agile Unified Process (AUP)**

The Agile Unified Process (AUP) adopts a "serial in the large" and "iterative in the small" philosophy for building computer-based systems. By adopting the classic UP phased activities—inception, elaboration, construction, and transition—AUP provides a serial overlay that enables a team to visualize the overall process flow for a software project. However, within each of the activities, the team iterates to achieve agility and to deliver meaningful software increments to end users as rapidly as possible. Each AUP iteration addresses the following activities.

• **Modeling**. UML representations of the business and problem domains are created.

• **Implementation**. Models are translated into source code.

• **Testing**. Like XP, the team designs and executes a series of tests to uncover errors and ensure that the source code meets its requirements.

• **Deployment**. Like the generic process activity deployment in this content focuses on the delivery of a software increment and the acquisition of feedback from end users.

• **Configuration and project management**. In the content of AUP, configuration management addresses change management, risk management, and the control of any persistent work products that are produced by the team. Project management tracks and controls the progress of the team and coordinates team activities.

• **Environment management**. Environment management coordinates a process infrastructure that includes standards, tools, and other support technology available to the team.

## A Tool set for the Agile process

The best way to do this is to deploy a set of good tools that help track the project and organize the team's progress. They don't impose strict schedules and roles, but merely make it easier for the developers to self-manage and converge on their goals. There are dozens of software products designed to help managers set priorities and developers write code that addresses them. Some of these tools are designed to track different forms of development, including projects that are more centrally managed, but they are flexible enough to be used for agile development. A common feature of all these agile tools is a graphic dashboard that reports how the team is progressing and meeting the goals.

# 1. Source control tools

**Git** is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency. It offers much of the flexibility that teams need to move ahead. The lack of one dominant central repository makes it simpler for different developers to follow different paths and then merge their code later. Git is widely supported, and many teams now use its hosting services to keep their code organized. Many of the other tools in this list take their cues from Git and use the updates to the repository to track and test progress. Other top source control tools include **Mercurial**, **Subversion**, and **CVS**.

# 2. Continuous integration tools

The tools automatically add a layer of processing when code is committed, helping to ensure that the team is working smoothly together. There are a number of good continuous integration tools that play well with agile management systems. Some of the best known tools include Hudson, Jenkins, Travis CI, Strider, and Integrity.

# 3. Team management tools

**Agile Manager**- HP's Agile Manager is built to organize and guide teams from the beginning as they plan and deploy working code through the agile model. At the early stages of the cycle during the release plan, the managers gather the user stories and decide how the teams will attack them. These set the stage for the sprints and deployment.

**Active Collab**-From juggling tasks to tracking time and generating bills, Active Collab is organized to help software shops deliver code and account for their time. The heart of the system is a list of tasks that can be assigned and tracked from conception to completion.

**JIRA Agile**- The JIRA Agile tool adds a layer for agile project management that interacts with the other major tools from Atlassian. The team creates a list of project tasks with a tool called Confluence and then tracks them on an interactive Kanban board that developers can update as they work.

**Agile Bench**-The Agile Bench tool is a hosted platform that emphasizes tracking the work assigned to each individual. The release schedule begins as a backlog of user stories and other enhancements. As they're assigned, the team must gauge both the business impact and the cost of development by assigning an estimate of the complexity of each task in points. The dashboard tracks both of these values so that members can tell who is overloaded and which tasks are the most important.

**Pivotal Tracker**- Pivotal Tracker is just one of a constellation of tools from Pivotal Labs created to support agile development. The core of the project is a page that lists the tasks that are often expressed as stories. Team members can rank the complexity with points, and the tool will track how many tasks are being finished each day.

**Telerik TeamPulse** - Telerik is known for its numerous frameworks for creating apps for the mobile marketplace. They've bundled much of that experience from creating their own code into TeamPulse, a tool they use to track projects.

**Version One**- Version One is designed to organize all the groups involved in development across an enterprise by providing a stable communication platform where everyone can plan the initiatives and create persistent documentation.

**Plan box** - Plan box offers four levels of organizational power to keep multiple teams working together toward a common goal. They contain projects, which are built on items that, in turn, are filled with tasks. As the team finishes the tasks, Planbox tracks the progress on all these levels and produces reports for all stakeholders.

**Lean Kit** – Learn Kit aims to imitate the conference room whiteboards where most projects begin. It lets all team members post virtual notes or cards that represent all the tasks, user stories, or bugs that must be addressed. As the team finishes them, the board

updates faster than any whiteboard. The software also allows multiple teams to work together in separate spaces while still coordinating their interactions.

**Axosoft**- Axosoft project tool tracks the project in three different ways. The Release Planner offers a tabular view of the different tasks, bugs, and user stories. Developers drag and drop the different entries to assign them and mark them as finished. The burndown charts show graphically how quickly the team is converging on its goal.

## Core principles of Software engineering

Principles that guide process Following set of core principles can be applied to every software process:

1. Be agile Principle
2. Be ready to adapt Principle
3. Focus on quality at every step Principle
4. Build an effective team Principle
5. Establish mechanisms for communication and coordination Principle
6. Manage change Principle
7. Assess risk Principle
8. Create work products that provide value for oath

## Principles that guide each framework activity

**Communication Principles**

1. Listen Principle
2. Prepare before you communicate Principle
3. Someone should facilitate the activity Principle
4. Face to face communication is best Principle
5. Take notes and document decisions Principle
6. Strive for collaboration Principle
7. Stay focused; modularize your discussion Principle
8. If something is unclear, draw a picture Principle
9. (a) Once you agree to something, move on (b) If you can't

**Planning Principles**

1. Listen Principle
2. Prepare before you Planning the Principle
3. Know about the active sessions of the Principle
4. Scheduling the tasks into simple is the best Principle
5. Monitor the tasks in proper manner.
6. Control the tasks in good manner.
7. See the planning framework performed correctly or not.
8. If planning deviates apply the necessary steps to order planning
9. Take the notes and document each step in planning principle

**Modeling Principles:**

1. Listen Principle
2. Prepare before you design the Principle.
3. Know about the active diagrams used in Principle.
4. For the functional programming use the structural design.
5. Object Oriented design is the best design in modeling principle.
6. Draw the Class, Object, Software and Hardware like static diagrams if required.
7. Draw the Use case, Sequence, Collaboration and Active like dynamic diagrams also if required.
8. Design in correct manner for proper coding.
9. Take the notes and document each step in Modeling principle.

**Construction Principles:**

1. Listen Principle
2. Prepare before you apply construction principle.
3. Know about the Importance of programming.
4. For structural design use the functional programming.
5. Object Oriented Programming used for the Object oriented design.
6. Use the GUI programming for webapps.
7. Follow the design in correct manner leads to good programming.
8. If developer misleaded by the design, cause to the programming bugs.
9. Take the notes and document each step in Construction principle.

**Deployment Principles:**

1. Listen Principle
2. Prepare before you apply Deployment principle.
3. Know about the product release principles.
4. Apply the testing on the developed coding.
5. If errors identified resolve the errors.
6. Release the product prototype iteration cycle by cycle.
7. Involve the customer in every iteration to get the more transparency.
8. If errors are decreased to customer satisfied level release the final product.
9. Take the notes and document each step in Deployment principle.

## REQUIREMENTS ENGINEERING

Requirements analysis, also called requirements engineering, is the process of determining user expectations for a new or modified product. Requirements engineering is a major software engineering action that begins during the communication activity and continues into the modeling activity. It must be adapted to the needs of the process, the project, the product, and the people doing the work. Requirements engineering builds a bridge to design and construction.

Requirements engineering provides the appropriate mechanism for understanding what the customer wants, analyzing need, assessing feasibility, negotiating a reasonable solution, specifying the solution unambiguously, validating the specification, and managing the requirements as they are transformed into an operational system. It encompasses seven distinct tasks: inception, elicitation, elaboration, negotiation, specification, validation, and management.

**Inception :** It establish a basic understanding of the problem, the people who want a solution, the nature of the solution that is desired, and the effectiveness of preliminary communication and collaboration between the other stakeholders and the software team.

**Elicitation:** In this stage, proper information is extracted to prepare to document the requirements. It certainly seems simple enough—ask the customer, the users, and others what the objectives for the system or product are, what is to be accomplished, how the system or product fits into the needs of the business, and finally, how the system or product is to be used on a day-to-day basis.

**Elaboration:** The information obtained from the customer during inception and elicitation is expanded and refined during elaboration. This task focuses on developing a refined requirements model that identifies various aspects of software function, behavior, and information. Elaboration is driven by the creation and refinement of user scenarios that describe how the end user (and other actors) will interact with the system.

**Negotiation:** To negotiate the requirements of a system to be developed, it is necessary to identify conflicts and to resolve those conflicts. You have to reconcile these conflicts through a process of negotiation. Customers, users, and other stakeholders are asked to rank requirements and then discuss conflicts in priority. Using an iterative approach that prioritizes requirements, assesses their cost and risk, and addresses internal conflicts, requirements are eliminated, combined, and/or modified so that each party achieves some measure of satisfaction.

**Specification:** The term specification means different things to different people. A specification can be a written document, a set of graphical models, a formal mathematical model, a collection of usage scenarios, a prototype, or any combination of these.

**Validation:** The work products produced as a consequence of requirements engineering are assessed for quality during a validation step. Requirements validation examines the specification to ensure that all software requirements have been stated unambiguously; that inconsistencies, omissions, and errors have been detected and corrected; and that the work products conform to the standards established for the process, the project, and the product.

The primary requirements validation mechanism is the technical review. The review team that validates requirements includes software engineers, customers, users, and other stakeholders who examine the specification looking for errors in content or interpretation, areas where clarification may be required, missing information, inconsistencies, conflicting requirements, or unrealistic requirements.

**Requirements management**. Requirements for computer-based systems change, and the desire to change requirements persists throughout the life of the system. Requirements management is a set of activities that help the project team identify, control, and track requirements and changes to requirements at any time as the project proceeds. Many of these activities are identical to the software configuration management (SCM) techniques.

## ESTABLISHING THE GROUNDWORK

**Identifying Stakeholders** - A stakeholder is anyone who has a direct interest in or benefits from the system that is to be developed. At inception, you should create a list of people who will contribute input as requirements are elicited.

**Recognizing Multiple Viewpoints** - Because many different stakeholders exist, the requirements of the system will be explored from many different points of view. The information

from multiple viewpoints is collected, emerging requirements may be inconsistent or may conflict with one another.

**Working toward Collaboration** - The job of a requirements engineer is to identify areas of commonality and areas of conflict or inconsistency. It is, of course, the latter category that presents a challenge. Collaboration does not necessarily mean that requirements are defined by committee. In many cases, stakeholders collaborate by providing their view of requirements, but a strong "project champion"(e.g., a business manager or a senior technologist) may make the final decision about which requirements make the cut.

**Asking the First Questions** - Questions asked at the inception of the project should be "context free" . The first set of context- free questions focuses on the customer and other stakeholders, the overall project goals and benefits. For example, you might ask:

- Who is behind the request for this work?
- Who will use the solution?
- What will be the economic benefit of a successful solution?
- Is there another source for the solution that you need?

These questions help to identify all stakeholders who will have interest in the software to be built. In addition, the questions identify the measurable benefit of a successful implementation and possible alternatives to custom software development. The next set of questions enables you to gain a better understanding of the problem and allows the customer to voice his or her perceptions about a solution:

- How would you characterize "good" output that would be generated by a successful solution?
- What problem(s) will this solution address?
- Can you show me (or describe) the business environment in which the solution will be used?
- Will special performance issues or constraints affect the way the solution is approached?

The final set of questions focuses on the effectiveness of the communication activity itself. Gause and Weinberg call these "meta-questions" and propose the following list:

- Are you the right person to answer these questions? Are your answers "official"?
- Are my questions relevant to the problem that you have?
- Am I asking too many questions?
- Can anyone else provide additional information?

• Should I be asking you anything else?

These questions will help to "break the ice" and initiate the communication that is essential to successful elicitation. But a question-and-answer meeting format is not an approach that has been overwhelmingly successful.

## ELICITING REQUIREMENTS

Requirements elicitation (also called requirements gathering) combines elements of problem solving, elaboration, negotiation, and specification.

**Collaborative Requirements Gathering** - Many different approaches to collaborative requirements gathering have been proposed. Each makes use of a slightly different scenario, but all apply some variation on the following basic guidelines:

• Meetings are conducted and attended by both software engineers and other stakeholders.

• Rules for preparation and participation are established.

• An agenda is suggested that is formal enough to cover all important points but informal enough to encourage the free flow of ideas.

• A "facilitator" (can be a customer, a developer, or an outsider) controls the meeting.

• A "definition mechanism" (can be work sheets, flip charts, or wall stickers oran electronic bulletin board, chat room, or virtual forum) is used.

The goal is to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of solution requirements in an atmosphere that is conducive to the accomplishment of the goal.

During inception basic questions and answers establish the scope of the problem and the overall perception of a solution. Out of these initial meetings, the developer and customers write a one- or two-page "product request." A meeting place, time, and date are selected; a facilitator is chosen; and attendees from the software team and other stakeholder organizations are invited to participate. The product request is distributed to all attendees before the meeting date.

**Quality function deployment** -(QFD) is a quality management technique that translates the needs of the customer into technical requirements for software. QFD "concentrates on maximizing customer satisfaction from the software engineering process". To accomplish this, QFD emphasizes an understanding of what is valuable to the customer and then deploys these values throughout the engineering process. QFD identifies three types of requirements :

• **Normal requirements**. -The objectives and goals that are stated for a product or system during meetings with the customer. If these requirements are present, the customer is satisfied.

Examples of normal requirements might be requested types of graphical displays, specific system functions, and defined levels of performance.

• **Expected requirements**: - These requirements are implicit to the product or system and may be so fundamental that the customer does not explicitly state them. Their absence will be a cause for significant dissatisfaction.

• **Exciting requirements**. -These features go beyond the customer's expectations and prove to be very satisfying when present. Although QFD concepts can be applied across the entire software process, QFD uses customer interviews and observation, surveys, and examination of historical data as raw data for the requirements gathering activity. These data are then translated into a table of requirements— called the customer voice table—that is reviewed with the customer and other stakeholders.

**Usage Scenarios** -As requirements are gathered, an overall vision of system functions and features begins to materialize. However, it is difficult to move into more technical software engineering activities until you understand how these functions and features will be used by different classes of end users. To accomplish this, developers and users can create a set of scenarios that identify a thread of usage for the system to be constructed. The scenarios, often called use cases, provide a description of how the system will be used.

**Elicitation Work Products**

The work products produced as a consequence of requirements elicitation will vary depending on the size of the system or product to be built. For most systems, the work products include

• A statement of need and feasibility.

• A bounded statement of scope for the system or product.

• A list of customers, users, and other stakeholders who participated in requirements elicitation.
• A description of the system's technical environment.

• A list of requirements and the domain constraints that apply to each.

• A set of usage scenarios that provide insight into the use of the system or product under different operating conditions.

• Any prototypes developed to better define requirements. Each of these work products is reviewed by all people who have participated in requirements elicitation.

## DEVELOPING USE CASES

Use cases are defined from an actor's point of view. An actor is a role that people (users) or devices play as they interact with the software. The first step in writing a use case is to define the set of "actors" that will be involved in the story. Actors are the different people (or devices) that use the system or product within the context of the function and behavior that is to be described.

Actors represent the roles that people (or devices) play as the system operates. Defined somewhat more formally, an actor is anything that communicates with the system or product and that is external to the system itself. Every actor has one or more goals when using the system. It is important to note that an actor and an end user are not necessarily the same thing. A typical user may play a number of different roles when using a system, whereas an actor represents a class of external entities (often, but not always, people) that play just one role in the context of the use case. Different people may play the role of each actor. Because requirements elicitation is an evolutionary activity, not all actors are identified during the first iteration. It is possible to identify primary actors during the first iteration and secondary actors as more is learned about the system.

Primary actors interact to achieve required system function and derive the intended benefit from the system. Secondary actors support the system so that primary actors can do their work. Once actors have been identified, use cases can be developed.

Jacobson suggests a number of questions that should be answered by a use case:

- Who is the primary actor, the secondary actor(s)?
- What are the actor's goals?
- What preconditions should exist before the story begins?
- What main tasks or functions are performed by the actor?
- What exceptions might be considered as the story is described?
- What variations in the actor's interaction are possible?
- What system information will the actor acquire, produce, or change?
- Will the actor have to inform the system about changes in the external environment?
- What information does the actor desire from the system?
- Does the actor wish to be informed about unexpected changes?

The basic use case presents a high-level story that describes the interaction between the actor and the system.

## BUILDING THE REQUIREMENTS MODEL

The intent of the analysis model is to provide a description of the required informational, functional, and behavioral domains for a computer-based system. The model changes dynamically as you learn more about the system to be built, and other stakeholders understand more about what they really require.

### Elements of the Requirements Model

The specific elements of the requirements model are dictated by the analysis modeling method that is to be used. However, a set of generic elements is common to most requirements models.

• **Scenario-based elements.** The system is described from the user's point of view using a scenario-based approach.

• **Class-based elements.** Each usage scenario implies a set of objects that are manipulated as an actor interacts with the system. These objects are categorized into classes—a collection of things that have similar attributes and common behaviors.

• **Behavioral elements.** The behavior of a computer-based system can have a profound effect on the design that is chosen and the implementation approach that is applied. Therefore, the requirements model must provide modeling elements that depict behavior.

• **Flow-oriented elements.** Information is transformed as it flows through a computer- based system. The system accepts input in a variety of forms, applies functions to transform it, and produces output in a variety of forms.

### Analysis Patterns

Analysis patterns suggest solutions (e.g., a class, a function, a behavior) within the application domain that can be reused when modeling many applications.Geyer-Schulz and Hahsler suggest two benefits that can be associated with the use of analysis patterns:

**First,** analysis patterns speed up the development of abstract analysis models that capture the main requirements of the concrete problem by providing reusable analysis models with examples as well as a description of advantages and limitations.

**Second,** analysis patterns facilitate the transformation of the analysis model into a design model by suggesting design patterns and reliable solutions for common problems.

Analysis patterns are integrated into the analysis model by reference to the pattern name.

## NEGOTIATING REQUIREMENTS

The intent of negotiation is to develop a project plan that meets stakeholder needs while at the same time reflecting the real-world constraints (e.g., time, people, budget) that have been placed on the software team. The best negotiations strive for a "win-win" result. That is, stakeholders win by getting the system or product that satisfies the majority of their needs and you win by working to realistic and achievable budgets and deadlines.

Boehm defines a set of negotiation activities at the beginning of each software process iteration. Rather than a single customer communication activity, the following activities are defined:

1. Identification of the system or subsystem's key stakeholders.

2. Determination of the stakeholders' "win conditions."

3. Negotiation of the stakeholders' win conditions to reconcile them into a set of win-win conditions for all concerned.

Successful completion of these initial steps achieves a win-win result, which becomes the key criterion for proceeding to subsequent software engineering activities.

## VALIDATING REQUIREMENTS

As each element of the requirements model is created, it is examined for inconsistency, omissions, and ambiguity. The requirements represented by the model are prioritized by the stakeholders and grouped within requirements packages that will be implemented as software increments.

A review of the requirements model addresses the following questions:

• Is each requirement consistent with the overall objectives for the system/product?

• 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?

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

• Is each requirement bounded and unambiguous?

• Does each requirement have attribution?

That is, is a source (generally, a specific individual) noted for each requirement?

• Do any requirements conflict with other requirements?

• Is each requirement achievable in the technical environment that will house the system or product?

• Is each requirement testable, once implemented?

• Does the requirements model properly reflect the information, function, and behavior of the system to be built?

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

• 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 stakeholder needs and that it provides a solid foundation for design.