3.1 Agile Software Development

Agile software engineering combines a philosophy and a set of development guidelines ensuring more customer satisfaction and early incremental delivery of software (Pressman, 2006). The characteristics of the principle are informal methods; minimal software engineering work products; small, highly motivated project teams; and overall development simplicity. The principle stress delivery over analysis and design through active and continuous communication between developers and customers. Software engineers and all stakeholders work together on an agile team - a team that is self-organizing and in control of its own destiny. An agile team fosters communication and collaboration among all who serve on it. It is important because the modern business environment that spawns computer-based system and software products is fast-paced and ever-changing. Agile software engineering represented a reasonable alternative to conventional software engineering for certain classes of software and certain types of software projects. It has been demonstrated to deliver successful systems quickly.

The basic framework activities are: communication, planning, modeling, construction and deployment. But they group into a minimal task set that pushes the project team toward construction and delivery. When both the customer and the software engineer agree that the "work product" is an operational, incremental software - it is delivered to the customer on the appropriate commitment date.

Beck et al. (2001a), (16 noted software developers, writers, and consultant) signed the "Manifesto for Agile Software Development", as stated below:

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- Individual and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.
A manifesto is normally associated with an emerging political movement—one that attacks the old guards and suggests revolutionary change. In some ways, that’s exactly what agile development is all about.

3.1.1 What is Agility?

In the context of software engineering work Ivar Jacobson (Jacobson, 2002) provides a useful discussion:

"Agility has become today’s buzzword when describing a modern software process. Everyone is agile. An agile team is a nimble team able to appropriately respond to changes 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”

But agility is more than an effective response to change. It emphasizes rapid delivery of operational software and de-emphasizes the importance of intermediate work products; it adopts the customer as a part of the development team and works to eliminate the "us and them" attitude that continues to pervade many software projects; it recognizes that planning in an uncertain world has its limits and that a project plan must be flexible.

Agility can be applied to any software process. However, to accomplish this, it is essential that the process be designed in a way that allows the project team to adapt tasks and to streamline them, conduct planning in way that understands the fluidity of an agile development approach, eliminate all but the most essential work products and keep them lean, and emphasize an incremental delivery strategy that gets working software to the customer as rapidly as feasible for the product type and operational environment.

Any agile software process is characterized in a manner that addresses a number of key assumptions (Fowler, 2002) 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 as we might like. Given these three assumptions, the question of unpredictability arises. Therefore, the development process must be adaptable to the rapidly changing scenarios of the project. An agile process, therefore, must be adaptable. But continual adaptation without forward progress accomplishes little. Therefore, an agile software process must adapt incrementally. To accomplish incremental adaptation, an agile team requires customer feedback.

Hence, an incremental development strategy should be instituted. Software increments must be delivered in short time periods so that adaptation keeps pace with change (unpredictability). This iterative approach enables the customer to evaluate the software increment regularly, provide necessary feedback to the software team, and influence the process adaptations that are made to accommodate the feedback.

3.1.2 Agility and the Cost of Change

The conventional wisdom in software development is that the cost of change increases non-linearly as a project progresses (Pressman, 2006). The cost of change as a function of development schedule progress is shown in figure 3.1. It is relatively easy to accommodate a change when a software team is gathering requirements. A Usage scenario might have to be modified, a list of functions may be extended or a written specification can be edited. The costs of doing this work are minimal and the time required will not adversely affect the outcome of the project. But the cost of change in the middle of validation testing shall be very high as a major functional change requires a modification to the architectural design of the software, three new components must be designed and constructed, modifications to another five components must be made, new tests must be designed and so on. Costs escalate quickly and the time and cost required to ensure that the change is made without unintended side effects is nontrivial.
3.1.3 Agility Principles

A set of 12 agility principles has been developed by the Agile Alliance (Anonymous, 2003; Fowler and Highsmith, 2001) to establish a basis for agile process model. Glen Alleman (2002) presents each of these principles and then considers the domain in which the principles are applicable and issues associated with their applicability within that domain.

1. The highest priority is to satisfy the customer through early and continuous delivery of valuable software. The concept of customer value for work performed is well developed in any business domain. However definitions of value, early, and satisfactions are not provided, so domain specific definitions need to be developed before this principle can be of practical use in a specific circumstance.

2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage. Agile process contributes to success in these situation. This might be considered the operational definition of agility. Early and repetitive feedback on product or project design is good practice in many engineering disciplines. But changing requirements may be an indication of changing business values, unstable requirements, or a lack of understanding of the desired business outcome. Without stable business success metrics, the creation of software to address unstable requirement is not good business strategy. A close examination of why these requirements are changing is
an important risk assessment step in determining if agile process will be successful. Late
detail binding and separation of concerns can support changing requirements; however,
decisions must still be made to identify the areas that require flexibility to deal with changing
requirement

3. Deliver working software frequently, from a couple of weeks to a couple of months, with a
preference to the shorter timescale. Agility helps here, but in general, iterative and
incremental methods exhibit this as a behavior, including Spiral method—without the
necessary re-labeling of agile. The concept of rapid prototyping is standard practice in many
manufacturing and engineering processes. The granularity of the deliverables is the issue
here. The question is – what is the appropriate absorption rate of the software iteration for a
specific domain?

4. Business people and developers must work together daily throughout the project. This is
common business practice in successful organizations. The definition of the customer is
restrictive in many of the agile process methods, especially when building products rather
than projects. The granularity of the interaction is the issue. If the customer is co-located,
direct daily interaction is possible. If not, then some other form of communication is
necessary. Documentation then plays a more significant role.

5. Build project around motivated individuals. Give them the environment and support they
need, and trust them to get the job done. This is common business practice in successful
organizations- it’s the people that make a project successful.

6. The most efficient and effective method of conveying information to and within a
development team is face-to-face conversation. Although this is the basis of agile
processes, this is neither unique nor many times practical and in many cases may not even
be desirable. Written specifications are useful in many instances and in others instances they
imposed by contract, geography, regulatory, or safety requirements. This principle is a
tautology, but provides no suggestions for alternatives.

7. Working software is the primary measure of progress. Although working software is an
outcome of development, there are other critical deliverables and measures of progress as
well that are not addressed by many opportunities for process improvement prior to and after
the generation of code.
8. **Agile processes promote sustainable development.** Although a goal, the agile process has little to say on how to achieve this in practice for a specific environment. As well, the statement on sustainability is a conjecture not yet supported by field evidence.

9. **Continuous attention to technical excellence and good design enhances agility.** This is the basis of many good engineering practices. The metrics of technical excellence and good design are not stated, leaving them open to interpretation.

10. **Simplicity – the art of maximizing the amount of work not done—is essential.** Without a context, the term simplicity has no meaning. What is simple in one domain may appear complex when viewed from another. This principle fails to address non-functional requirements of product and project based processes that are the sources of much of the complexity in large scale systems.

11. **The best architecture, requirements, and designs emerge from self-organizing teams.** This is conjecture and is not based on analytical measurements. This principle does not state the domains in which it is applicable. The science of systems engineering has much to say here, but no recognition to this previous work is provided.

12. **At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.** This is good team development practice independent of the software environment. No metrics are provided by which to assess past behavior for adjust of future behavior.

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

### 3.1.4 Agility in Software Development Industry

Being proactive has a definite edge over being reactive. Proactive means setting the pace for others to "catch-up" and "reactive" is exactly the opposite. Reactive enterprises often find themselves scampering to meet the challenges set by an innovative and aware enterprise. The level of awareness and the I.Q. would determine whether an enterprise is proactive or reactive (Kar, 2006). The relationship between awareness, I.Q. and agility determine the state of an enterprise which is shown in figure 3.2. As the level of awareness and I.Q. increases, the agility also increases making the enterprise more proactive.
Agility in an enterprise can be attained using a simple two step approach (Kar, 2006):

**Step 1: Handling external factors**

The external environment for an enterprise includes all the factors that directly or indirectly impact business operations. Figure 3.3 shows the factors that influence an enterprise. Some of these factors are:

**Market**: The market conditions are generally volatile and change dramatically. The reasons for change might be totally unrelated to the industry. An agile enterprise is not caught unaware by the rapidly changing market scenario. It embraces the change and devises ways to use it to its own advantage. Sometimes, an agile enterprise introduces innovations that change the market scenario. This results in the realignment of the business plans of its competitors.

**Government Regulations**: New laws could be enacted or old ones modified over time that may impact the business. Usually such changes are notified well in advance by the regulatory authorities. Awareness is the key with respect to this factor.

**Customer**: The customer role has gone through rapid changes in recent times from being a mere end user to playing a more active role in the development process for the product leading to “co-creation of value”. An agile enterprise welcomes the customer feedback and tries to incorporate them in the product. It also strives to foresee some of the changing business needs and provides suggestion for the same to the customer. This adds value to provider-customer relationship and opens new avenues for future business.

---

**Fig. 3.2: Relationship between Awareness, I.Q. and Agility**

---

34
Natural Calamities: Natural calamities can be dealt with by having a business continuity plan and appropriate disaster recovery initiatives to prevent a complete shutdown operations and ensure that critical activities can keep on running in face of such situations.

Step 2: Formulating Internal Policies
The business enterprise is a synergy of various smaller units performing different functions. Agility at the enterprise level can be achieved when all these units themselves become agile. These can be achieved by formulating policies to govern the functions of the different units. It is very important to formulate flexible policies that evolves over time and a policy that acts as an enabler and not as a limiting factor.

3.2 Extreme Programming (XP) approach of Agile Development

Agile software development aims at fast communication and incremental delivering of software artifacts. Several approaches have been proposed and largely used in practice such as:

- Extreme Programming (XP) (Beck, 1999; Beck and Andreas, 2004b),
- Crystal Methodologies (Cockburn, 2002a),
Scrum (Schwaber, 1995; Anonymous, 1996),
Adaptive Software Development (ASD) (Highsmith, 1998),
Feature Driven Development (FDD) (Coad et al., 1999),
Agile Modeling (AM) (Ambler, 2002),
Dynamic Systems Development Method (DSDM) (Stapleton, 1997).

Each model adopts a subtly different approach to the agility problem. Within each model there is a set of "ideas" that represent a significant departure from traditional software engineering. And yet, many agile concepts are simply adaptations of good software engineering concepts.

Extreme Programming (XP) is the most widely used approach to agile software development. Although early work on the ideas and methods associated with Extreme Programming occurred during the late 1980s, the seminal work on the subject, has been written by Beck (1999).

3.2.1 XP Values

Beck (1999) 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 (verbal) collaboration between customers and developers, the establishment of effective metaphors for communicating importance 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 refactored 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 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. As an increment is delivered to a customer, the user stories or use cases that are implemented by the increments are used to perform acceptance tests. The degree to which the software implements the output, function and
behavior of the use case is a form of feedback. Finally, as new requirements are derived as part of iterative planning, the team provides the customer with rapid feedback regarding cost and schedule impact.

Beck and Andreas (2004b) argue that strict adherence to certain XP practices demands discipline. For example, there is often significant pressure to design for future requirements. Most software teams succumb, arguing that "designing for tomorrow" will save time and effort in the long run. An agile XP team must have the discipline to design for today, recognizing that future requirements may change dramatically, thereby demanding substantial rework of the design and implemented code.

By following the values, the agile team inculcates respect among its 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.

3.2.2 The XP Process

Extreme Programming uses an object-oriented approach as its preferred development paradigm and encompasses a set of rules and practices that occur within the context of four framework activities: planning, design, coding, and testing. Figure 3.4 illustrates the XP process and notes some of the key ideas and tasks that are associated with each framework activity. Key XP activities are summarized in the paragraphs that follows.

3.2.2.1 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. Listening leads to the creation of a set of "stories" (also called user stories) that describes required output, features, and functionality for software to be built. Each story is written by the customer and is placed on an index card. The customer assigns a value (i.e., a priority) to the story based on the overall business value of the feature or function. Members of the XP team then assess each story and assign a cost—measured in development weeks—to it. If the story is estimated to require more than 3 development weeks, the customer is asked to split the story into smaller stories and the assignment of
value and cost occurs again. It is important to note that new stories can be written at any time.

Customers and developers work together to decide how to group stories into the next release (the next software increment) to be developed by the XP team. Once a basic commitment (agreement on stories to be included, delivery date and other project matters) is made for a release, the XP team orders the stories that will be developed in one of three ways:

1. all stories will be implemented immediately (within a few weeks);
2. the stories with highest value will be moved up in the schedule and implemented first,
3. the riskiest stories will be moved up in the schedule and implemented first.

After the first project release (also called a small release of software increment) has been delivered, the XP team computes project velocity. Project velocity is the number of customer stories implemented during the first release and can then be used to:
(1) help estimate delivery dates and schedule for subsequent releases, and
(2) determine whether an over-commitment has been made for all stories across the
entire development project.

If an over-commitment occurs, the content of releases is modified or end delivery dates are
changed. As development work proceeds, the customer can add stories, change the value of
an existing story, split stories, or eliminate them. The XP team then reconsiders all remaining
releases and modifies its plans accordingly.

3.2.2.2 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 is discouraged.

XP encourages the use of CRC (class-responsibility-collaborator) cards (Beck, 1999) as an
effective mechanism for thinking about the software in an object-oriented context. CRC
cards identify and organize the object-oriented classes that are relevant to the current
software increment. The CRC cards are the only design work product produced as part of the
XP process. 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. The intent is to
lower the risk when true when true implementation starts and to validate the original
estimates for the story, containing the design problem. XP encourages refactoring — a
construction technique that is also a design technique. Fowler (2000) 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 modify/simplify the internal design that minimizes the chances of
introducing bugs. In essence, when you refactor you are improving the design of the
code after it has been written.”

3.2.2.3 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 (software increment). Once the unit test has been created, the developer is better able to focus on what must be implemented to pass the test. Nothing extraneous is added i.e. KIS. 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 (the code is reviewed as it is created). It also keeps the developers focused on the problem at hand. In practice, each person takes on a slightly different role. For example, one person might think about the coding details of a particular portion of the design while the other ensures that coding standards are being followed or that the code for the story will satisfy the unit test that has been developed to validate the code against the story.

As pair programmers complete their work, the code they develop is integrated with the work of others. In some cases this is performed on a daily basis by an integration team. In other cases, the pair programmers have integration responsibility. This "continuous integration" strategy helps to avoid compatibility and interfacing problems and provides a "smoke testing" environment that helps to uncover errors early.

3.2.2.4 Testing
We have already noted that the creation of unit test 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.

As the individual unit tests are organized into a "universal testing suite" (Wells, 1999), integration and validation testing of the system can occur on a daily basis. This provides the XP team with a continual indication of progress and also can raise warning flags early if things go awry. Wells (1999) states:

"Fixing small problems every few hours take less time than fixing huge problems just before the deadline."

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

3.3 SCRUM Approach of Agile Development

SCRUM treats major portions of systems development as a controlled black box (Schwaber, 1995). Concepts from industrial process control are applied to the field of systems development. Industrial process control defines processes as either “theoretical” (fully defined) or “empirical” (black box). When a black box process is treated as a fully defined process, unpredictable results occur.

A significant number of systems development processes are not completely defined, but are treated as though they are (Brooks, 1987; Booch, 1994). Unpredictability without control results. The SCRUM approach treats these systems development processes as a controlled black box.

The SCRUM methodology, (called after the SCRUM in rugby (Noyes, 2002) —a group responsible for picking up the ball and moving it forward) is a management, enhancement and maintenance methodology for an existing system initially documented by Pittman and Matthew (1993) and later expanded upon by Booch (1995). In SCRUM, software product releases are planned based on the following variables (Schwaber, 1995):

- Customer requirements-how the current system needs enhancing.
- Time pressure-what time frame is required to gain a competitive advantage.
- Competition-what is the required quality, given the above variables.
- Vision-what changes are required at this stage to fulfill the system vision.
- Resource-what staff and funding are available.

These variables form the initial plan for a software enhancement project. However, these variables also change during the project. A successful development methodology must take these variables and their evolutionary nature into account.

Systems are developed in a highly complicated environment. The complexity is both within the development environment and the target environment. Schwaber and Beedle present some of the major environmental variables (Schwaber and Beedle, 2001):

- Availability of skilled professionals-the newer the technology, tools, methods, and domain, the smaller the pool of skilled professionals.
• Stability of implementation technology - the newer the technology, the lower the stability and the greater the need to balance the technology with other technologies and manual procedures.

• Stability of power of tools - the newer and more powerful the development tool, the smaller the pool of skilled professionals and the more unstable the tool functionality.

• Effectiveness of methods - what modeling, testing, version control, and design methods are going to be used and how effective, efficient, and proven are they.

• Domain expertise - are skilled professionals available in the various domains, including business and technology.

• New feature - what entirely new features are going to be added, and to what degree will these fit with current functionality.

• Methodology - does the overall approach to developing systems and using the selected methods promote flexibility, or is this rigid, detailed approach that restricts flexibility.

• Competition - what will the competition do during the project? What new functionality will be announced or released.

• Time/Funding - how much time is available initially and as the project progresses? How much development funding is available?

• Other variables - any other factors that must be responded to during the project to ensure the success of the resulting, delivered system, such as reorganizations.

Schwaber and Beedle (2001) define the overall complexity as a function of these variables:

\[ \text{Complexity} = f(\text{development environment variables and target environment variables}) \]

As the complexity of the project increases, the greater is the need for controls, particularly the ongoing assessment and response to tasks. Figure 3.5 shows the poor probability success of waterfall model for complex projects. However, the success rate is high for the simple type of projects. Figure 3.6 shows the high probability success of simple as well as complex projects using agile model.
Inflexible response to unpredictability (internal & external) causes sharp drop in $p(\text{Success})$ as complexity increases.

Fig. 3.5: Probability graph for success of projects using waterfall model.

Flexible response to unpredictability improves $p(\text{Success})$ to Complexity relationship.

Fig. 3.6: Probability graph for success of projects using agile model.
3.3.1 Scrum Methodology

The characteristics of SCRUM methodology are (Schwaber, 1995):

- The first and the last phases consist of defined processes, where all processes, inputs and outputs are well defined. The knowledge of how to do these processes is explicit. The flow is linear, with some iterations in the planning phase.
- The sprint phase is an empirical process. Many of the processes in the sprint phase are unidentified or uncontrolled. It is treated as a black box that requires external controls. Accordingly, controls including risk management are put on each iteration of the sprint phase to avoid chaos while maximizing flexibility.
- Sprints are non-linear and flexible. Where available, explicit process knowledge is used; otherwise tacit knowledge and trial and error is used to build process knowledge. Sprints are used to evolve the final product.
- The project is open to the environment until the closure phase. The deliverable can be changed at any time during the planning and sprint phases of the project. The project remains open to the environmental complexity, including competitive, time, quality, and financial pressures, throughout these phases.
- The deliverable is determined during the project based on environment.

The three SCRUM phases (fig. 3.7) as defined by Schwaber (1995) is given below:

![Scrum Methodology Diagram](image-url)

Fig: 3.7 Various phases in Scrum Methodology
**Pre-game Phase**

*Planning:* Definition of a new release based on currently known backlog, along with an estimate of its schedule and cost. If a new system is being developed, this phase consists of both conceptualization and analysis. If an existing system is being enhanced, this phase consists of limited analysis.

*Architecture:* Design how the backlog items will be implemented. This phase includes system architecture modification and high level design.

**Game Phase**

*Development Sprints:* Development of new release functionality, with constant respect to the variables of time, requirements, quality, cost, and competition. Interaction with these variables defines the end of this phase. There are multiple, iterative development sprints, or cycles, that are used to evolve the system.

**Postgame Phase**

*Closure:* Preparation for release, including final documentation, pre-release staged testing, and release. When the management team feels that the variables of time, competition, requirements, cost, and quality concur for a new release to occur, they declare the release “closed” and enter this phase. This phase prepares the developed product for general release. Integration, system test, user documentation, training material preparation, and marketing material preparation are among closure tasks.

### 3.3.2 Controls in Scrum

Operating at the edge of unpredictability and complexity requires management controls to avoid falling into chaos (Anonymous, 1996a). The SCRUM methodology embodies these general, loose controls, using Object Oriented techniques for the actual construction of deliverables. Risk is the primary control. Risk assessment leads to changes in other controls and responses by the team. The controls in the SCRUM as stated by Schwaber and Beedle (2001) are *Backlog, Release/Enhancements, Packets, Changes, Problems, Risks, Solutions and Issues*. These controls are used in the various phases of SCRUM. Management uses these controls to manage backlog. Teams use these controls to manage changes, problems. Both management and teams jointly manage issues, risks, and solutions. These controls are reviewed, modified, and reconciled at every Sprint review meeting.
3.3.3 Deliverables of Scrum

The delivered product is flexible. Its content is determined by environment variables, including time, competition, cost, or functionality. The deliverable determinants are market intelligence, customer contact, and the skill of developers. Frequent adjustments to deliverable content occur during the project in response to environment. The deliverable can be determined anytime during the project.

3.3.4 Scrum Project Team

Variants of the SCRUM approach for new product development with high performance small teams was first observed by Takeuchi and Nonaka (1986) at Fuji –Xerox, Canon, Honda, NEC, Epson, Brother, 3M, Xerox, and Hewlett-Packard. The SCRUM team that works on the new release includes full time developers and external parties who will be affected by the new release, such as marketing, sales, and customers. In traditional release processes, these latter groups are kept away from development teams for fear of over-complicating the process and providing “unnecessary” interference. The SCRUM approach however welcomes and facilitates their controlled involvement at set intervals, as this increases the probability that release content and timing will be appropriate, useful, and marketable.

The following teams are formed for each new release:

- **Management**: Led by the Product Manager, it defines initial content and timing of the release, then manages their evolution as the project progresses and variables change. Management deals with backlog, risk, and release content.
- **Development teams**: Development teams are small, with each containing developers, documenters and quality control staff. One or more teams of between three and six people each are used. Each is assigned a set of packets (or objects), including all backlog items related to each packet. The team defines changes required to implement the backlog items in the packets, and manages all problems regarding the changes. Teams can be either functionally derived (assigned those packets that address specific sets of product functionality) or system derived (assigned unique layers of the system).

3.3.5 Advantages of Scrum

Additional development methodologies are designed any designed only to respond to the unpredictability of the external and the development environments at the start of an
enhancement cycle. Such newer approaches as the Boehm spiral methodology and its variants are still limited in their ability to respond to changing requirements once the project has started.

The SCRUM methodology, on the other hand, is designed to be quite flexible throughout. It provides control mechanisms for planning a product release and then managing variables as the project progresses. This enables organizations to change the project and deliverables the most appropriate release. The SCRUM methodology frees developers to devise the most ingenious solutions throughout the project, as learning occurs and the environment changes. Small, collaborative teams of developers are able to share tacit knowledge about development processes. An excellent training environment for all parties is provided. Object oriented technology provides the basis for the SCRUM methodology. Objects, or product features, offer a discrete and manageable environment. Procedural code, with its many and intertwined interfaces, is inappropriate for the SCRUM methodology. SCRUM may be selectively applied to the procedural systems with clean interfaces and strong data orientation.

3.4 The Impact of Agile Processes on Requirements Engineering

Agile software development aims at fast communication and incremental delivering of software artifacts. Several approaches have been proposed (refer to section 3.2) and largely used in practice. All agile processes resolve a paradox of requirements engineering –the need to formulate a clear vision of a system in a world of constantly changing requirements. For any system development, the degree of complexity increases as the requirements are less known and the technology is less certain. This is diagrammatically shown in figure 3.8.

Frequent inspection and immediate adaptation to the results are the primary mechanisms for dealing with system development complexity. Current agile practices are based on the following mechanisms:

1. Iterative Development – frequent iterations generate increments of work that can be inspected to determine the state of the project and serve as a basis for adaptation.

2. Increments of work – composed of working system functionality rather than artifacts. These increments create a 1:1 relationship between progress and product delivery, and provide a mechanism for user feedback to real product.

3. Collaboration – the customer and engineers form teams that work together.
Fig. 3.8: Degree of Complexity of RE for uncertain Technology

4. *Daily meetings* – provide a daily picture of the internal status of a project.
5. *Adaptation* – the teams of developers self-organize daily, based on the daily meetings and the developers and customers self-organize at the end of every increment to guide the project to create the greatest value.
6. *Emergence* – the architecture, team structure, and requirements emerge during the course of the project rather than being determined at the outset. The team is guided by preliminary and sketchy visions of requirements.

The fundamental principle underlying requirements engineering is the assumption that a system should be clearly specified before its design and implementation can start. Ideally a specification should be unambiguous, consistent, concise, traceable, implementation independent and signed off by all stakeholders involved before any further development is done.

3.5 Quick Review of Agile Software Development Literature

A quick review of the previous research work in the area of Agile Software Development, is presented in the following section:
<table>
<thead>
<tr>
<th>Year</th>
<th>Author(s)</th>
<th>Present (s)</th>
<th>Publisher/Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>1987</td>
<td>K, Beck &amp; W. Cunningham</td>
<td>Using Pattern Languages for Object-Oriented Programs</td>
<td>OOPSLA’87</td>
</tr>
<tr>
<td>1989</td>
<td>Kent Beck and W Cunningham</td>
<td>A Laboratory For Teaching Object-Oriented Thinking</td>
<td>OOPSLA’89</td>
</tr>
<tr>
<td>1989</td>
<td>Kent Beck</td>
<td>Simple Smalltalk Testing: With Patterns</td>
<td>Origins of xUnit frameworks</td>
</tr>
<tr>
<td>1996</td>
<td>Kent Beck</td>
<td>Smalltalk Best Practice Patterns</td>
<td>Prentice Hall</td>
</tr>
<tr>
<td>1997</td>
<td>M. Fowler</td>
<td>UML Distilled</td>
<td>Addison- Wesley</td>
</tr>
<tr>
<td>1999</td>
<td>Kent Beck</td>
<td>Extreme Programming Explained: Embrace Change.</td>
<td>Addison-Wesley (Winner of the Jolt Productivity Award)</td>
</tr>
<tr>
<td>Year</td>
<td>Author(s)</td>
<td>Present(s)</td>
<td>Publisher/Source</td>
</tr>
<tr>
<td>------</td>
<td>-----------</td>
<td>---------------------------------------------------------------------------</td>
<td>------------------------------------------------------</td>
</tr>
<tr>
<td>1999</td>
<td>P. Coad, E. Lefebvre, and J. DeLuca</td>
<td>Java Modelling in Color with UML.</td>
<td>Prentice-Hall</td>
</tr>
<tr>
<td>2000</td>
<td>Kent Beck and M. Fowler</td>
<td>Planning Extreme Programming Development</td>
<td>Addison-Wesley</td>
</tr>
<tr>
<td>2000</td>
<td>M. Fowler et al.</td>
<td>Refactoring: Improving the Design of Existing Code</td>
<td>Addison-Wesley</td>
</tr>
<tr>
<td>2001</td>
<td>K. Beck, and M. Fowler</td>
<td>Planning Extreme Programming Development</td>
<td>Addison-Wesley</td>
</tr>
<tr>
<td>1999</td>
<td>Kent Beck</td>
<td>Extreme Programming Explained: Embrace Change</td>
<td>Addison-Wesley</td>
</tr>
<tr>
<td>2002</td>
<td>M. Fowler</td>
<td>The New Methodology</td>
<td><a href="http://www.martinfowler.com/articles/newmethodologie">http://www.martinfowler.com/articles/newmethodologie</a> s.html #N8B</td>
</tr>
<tr>
<td>2002</td>
<td>A. Cockburn</td>
<td>Agile Software Development,</td>
<td>Addison-Wesley</td>
</tr>
<tr>
<td>Year</td>
<td>Author(s)</td>
<td>Present(s)</td>
<td>Publisher/Source</td>
</tr>
<tr>
<td>------</td>
<td>-----------</td>
<td>------------</td>
<td>------------------</td>
</tr>
<tr>
<td>2002</td>
<td>J. Highsmith</td>
<td>Agile Software Development Ecosystems</td>
<td>Addison-Wesley</td>
</tr>
<tr>
<td>2003</td>
<td>The Agile Alliance Group</td>
<td>The Agile Alliance Home Page,</td>
<td><a href="http://www.agilealliance.org/home">http://www.agilealliance.org/home</a></td>
</tr>
<tr>
<td>2004</td>
<td>Kent Beck</td>
<td>JUnit Pocket Guide.</td>
<td>O'Reilly</td>
</tr>
<tr>
<td>2004</td>
<td>A. Cockburn</td>
<td>Crystal Clear: A Human Powered Methodology for Small Teams</td>
<td>Addison-Wesley, 2004</td>
</tr>
<tr>
<td>2008</td>
<td>Kent Beck</td>
<td>Implementation Patterns</td>
<td>Addison-Wesley.</td>
</tr>
</tbody>
</table>