ATME COLLEGE OF ENGINEERING

13th KM Stone, Bannur Road, Mysore - 560 028

DEPARTMENT OF COMPUTER SCIENCE AND DESIGN

LECTURE NOTES

COURSE: SOFTWARE ENGINEERING & PROJECT MANAGEMENT

COURSE CODE: BCS501

SEMESTER: V

COURSE COORDINATOR: Mr. Chetan G

(ACADEMIC YEAR 2026-27)

MODULE 3

CHAPTER 1---AGILE DEVELOPMENT

1.1 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.

- The aim of agile process is to deliver the working model of software quickly to the customer For example: Extreme programming is the best known of agile process.

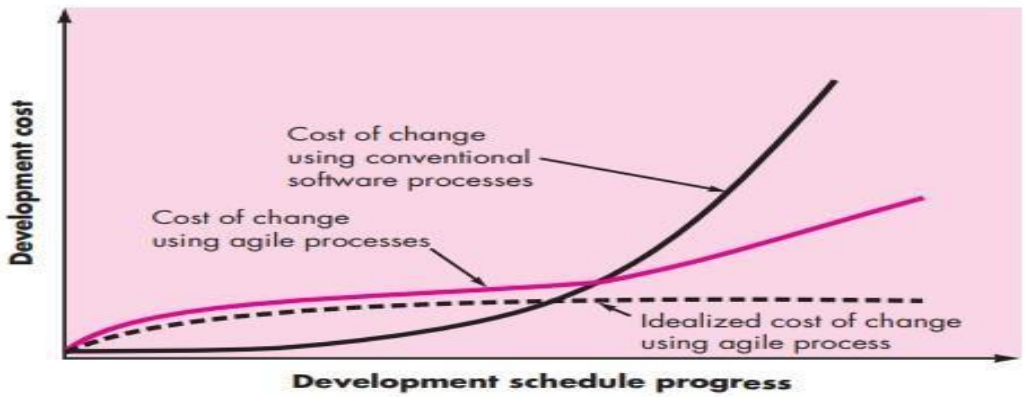
1.2 AGILITY AND THE COST OF CHANGE

- In conventional software development the cost of change increases non linearly as a project progresses (**Fig Solid Black curve).**

- **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 (**shaded, solid curve**), 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(reduced). 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. application, design, architecture etc. The verification process involves activities like reviews, walk-throughs and inspection.



1.3 WHAT IS AN 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.

Given these three assumptions, an important question arises:

1) How do we create a process that can manage unpredictability?

It lies in process **adaptability**. **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. An effective catalyst for customer feedback is an operational prototype or a portion of an operational system. Hence, an incremental development strategy should be instituted. Software increments must be delivered in short time periods so that adaptation keeps pace with change.

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.

1.3.1 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.

Not every agile process model applies these 12 principles with equal weight, and some models choose to ignore (or at least downplay) the importance of one or more of the principles.

1.3.2 The Politics of Agile Development

- There is debate about the benefits and applicability of agile software development as opposed to more conventional software engineering processes (produces documents rather than working product).

- Even within the agile, there are many proposed process models each with a different approach to the agility.

1.3.3 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.*

1. **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.

2. **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.

3. **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.

4. **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.

5. **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.

6. **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."

7. **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.

1.4 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 proposed by **Kent beck** during the late 1980's.

1.4.1 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 metaphors 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 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 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 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.

1.4.2 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**. Following figure illustrates the XP process and notes some of the key ideas and tasks that are associated with each framework activity.

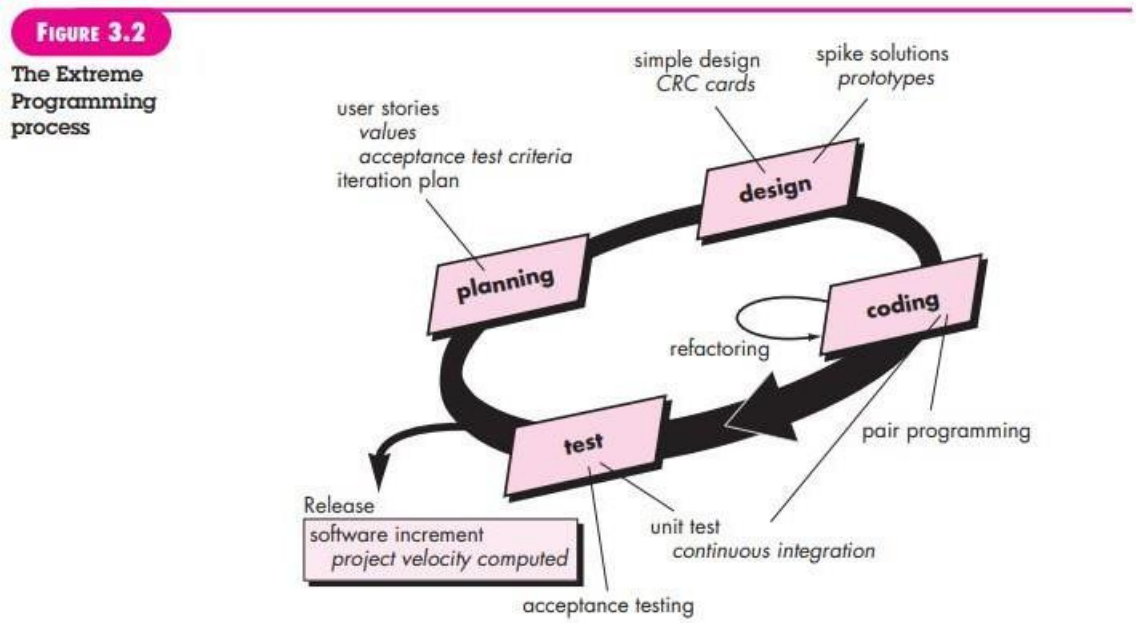


Fig : The Extreme Programming process

Key XP activities are

1) **Planning**. The planning activity begins with listening—a requirements gathering activity.

- **Listening** leads to the creation of a set of "stories" (also called user stories) that describe 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 three development weeks, 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.

- The stories with highest value will be moved up in the schedule and implemented first

2) **Design**. XP design rigorously follows the KIS (keep it simple) principle.

- 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.**

- XP encourages **refactoring**—a construction technique that is also a method for design optimization.

- Refactoring is the process of changing a software system in a way that it does not change the external behavior of the code and improves the internal structure.

3) **Coding**. After stories are developed and preliminary design work is done, the team does not move to code, develops a series of unit tests for 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.
- Once the code is complete, it can be unit-tested immediately, and providing feedback to the developers.
- A key concept during the coding activity is pair programming. i.e.., two people work together at one computer workstation to create code for a story.
- As pair programmers complete their work, the code they develop is integrated with the work of others.

4) **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.

- As the individual unit tests are organized into a "universal testing suite" 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 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.

1.4.3 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.

1.4.4 The XP Debate

- **Requirements volatility**. The customer is an active member of the XP team, changes to requirements are requested informally. As a consequence, the scope of the project can change and earlier work may have to be modified to accommodate current needs.

- **Conflicting customer needs.** Many projects have multiple customers, each with his own set of needs.

- **Requirements are expressed informally.** User stories and acceptance tests are the only explicit manifestation of requirements in XP. specification is often needed to remove inconsistencies, and errors before the system is built.

- **Lack of formal design:** when complex systems are built, design must have the overall structure of the software then it will exhibit quality.

1.5 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)
- Agile Unified Process (AUP)

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.

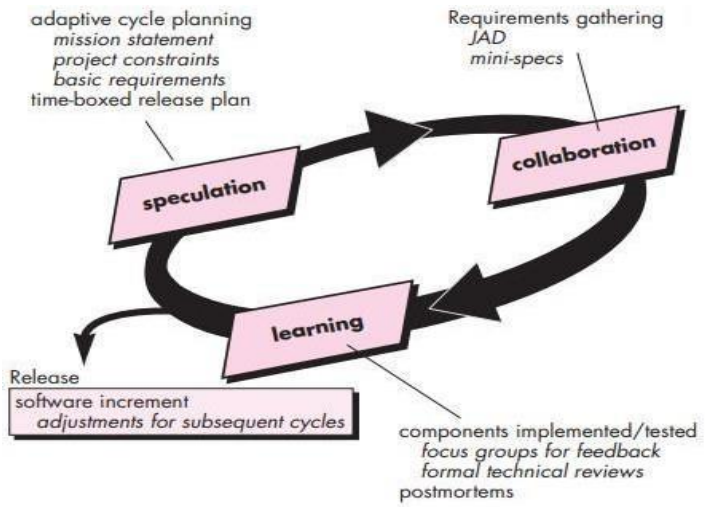


Fig: Adaptive software development

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.

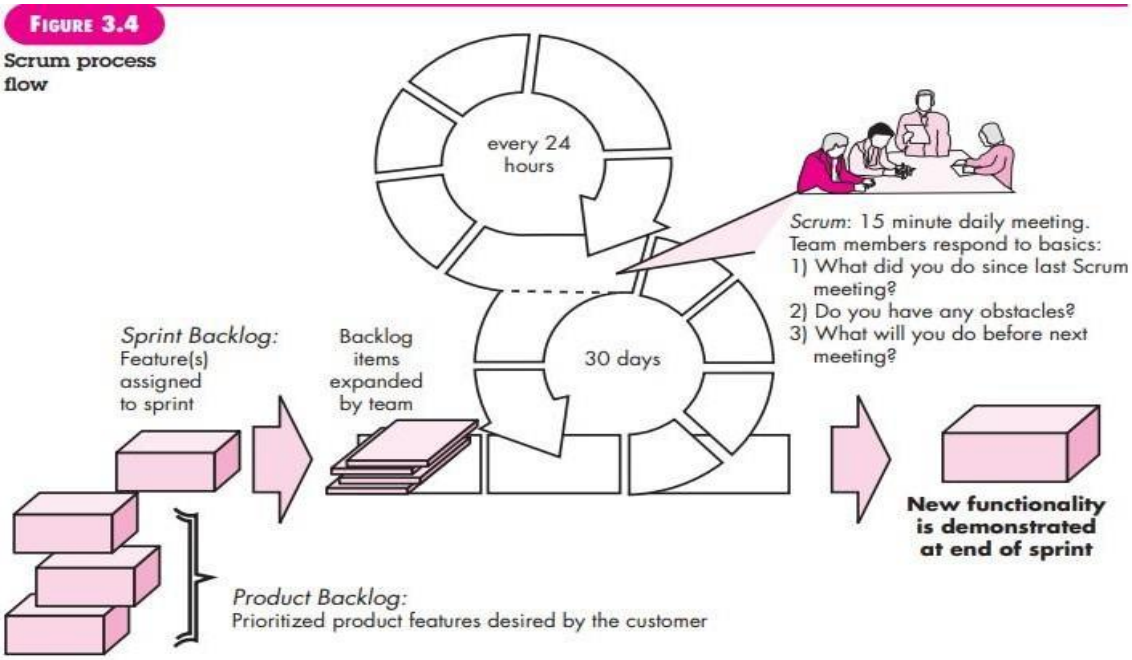
Motivated people use **collaboration** in a way that multiplies their talent and creative output beyond their absolute numbers. This approach is a recurring theme in all agile methods. But collaboration is not easy. It encompasses communication and teamwork, but it also emphasizes individualism, because 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

- What did you do since the last team meeting?
- What obstacles are you encountering?
- 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. That is, only enough work is required for each increment to facilitate movement to the next increment. The remaining detail can be completed later when more business requirements are known or changes have been requested and accommodated.

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. The intent during this iterative cycle is to gather additional requirements by eliciting feedback from users as they exercise the prototype.

- **Design and build iteration—**revisits prototypes built during functional model iteration to ensure that each has been engineered in a manner that will enable it to provide operational business value for end users. In some cases, functional model iteration and design and build iteration occur concurrently.

- **Implementation**—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 improved Coad'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.

FDD emphasizes software quality assurance activities by encouraging an incremental development strategy, the use of design and code inspections, the application of software quality assurance audits, the collection of metrics, and the use of patterns (for analysis, design, and construction).

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:

<**action**> the <**result**> < **by|for| of |to**> a(n) <**object**>
Where an <**object**> is "a person, place or thing.

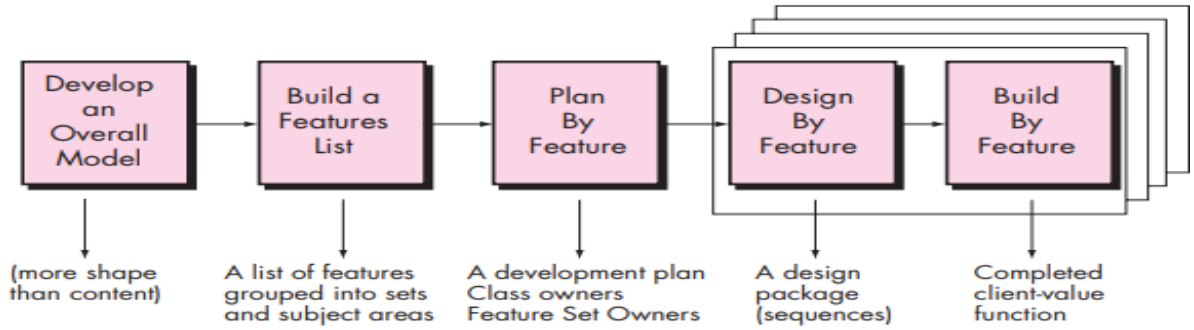


Fig: Feature Driven Development (FDD)

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.

For example, eliminate waste within the context of an agile software project as

1) adding no extraneous features or functions
(2) assessing the cost and schedule impact of any newly requested requirement,
(3) removing any superfluous process steps,
(4) establishing mechanisms to improve the way team members find information,
(5) ensuring the testing finds as many errors as possible,

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 adopts all of the values that are consistent with the agile manifesto. The agile modeling philosophy recognizes that an agile team must have the courage to make decisions that may cause it to reject a design and refactor. The team must also have the humility to recognize that technologists do not have all the answers and that business experts and other stakeholders should be respected and embraced.

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 context focuses on the delivery of a software increment and the acquisition of feedback from end users.

- **Configuration and project management.** In the context 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.

1.6 A TOOL SET FOR AGILE PROCESS

Some proponents of the agile philosophy argue that automated software tools (e.g., design tools) should be viewed as a minor supplement to the team's activities, and not at all pivotal to the success of the team.

However, Alistair Cockburn [Coc04] suggests that tools can have a benefit and that "agile teams stress using tools that permit the rapid flow of understanding. Some of those tools are social, starting even at the hiring stage. Some tools are technological, helping distributed teams simulate being physically present.

Many tools are physical, allowing people to manipulate them in workshops." Because acquiring the right people (hiring), team collaboration, stakeholder communication, and indirect management are key elements in virtually all agile process models, Cockburn argues that "tools" that address these issues are critical success factors for agility.

For example, a hiring "tool" might be the requirement to have a prospective team member spend a few hours pair programming with an existing member of the team. The "fit" can be assessed immediately.

Collaborative and communication "tools" are generally low tech and incorporate any mechanism ("physical proximity, whiteboards, poster sheets, index cards, and sticky notes" [Coc04]) that provides information and coordination among agile developers

Active communication is achieved via the team dynamics (e.g., pair programming), while passive communication is achieved by "information radiators" (e.g., a flat panel display that presents the overall status of different components of an increment).

Project management tools deemphasize the Gantt chart and replace it with earned value charts or "graphs of tests created versus passed . . . other agile tools are used to optimize the environment in which the agile team works (e.g., more efficient meeting areas), improve the team culture by nurturing social interactions (e.g., collocated teams), physical devices (e.g., electronic whiteboards), and process enhancement (e.g., pair programming or time-boxing)" [Coc04].

CHAPTER 2
PRINCIPLES THAT GUIDE THE PRACTICE

Practice is a collection of concepts, principles, methods, and tools that a software engineer calls upon on a daily basis.

Practice allows managers to manage software projects and software engineers to build computer programs.

Practice populates a software process model with the necessary technical and management how-to's to get the job done.

1.1 SOFTWARE ENGINEERING KNOWLEDGE

In an editorial published in IEEE Software a decade ago, Steve McConnell [McC99] made the following comment:

Many software practitioners think of software engineering knowledge almost exclusively as knowledge of specific technologies: Java, Perl, html, C++ Linux, Windows NT, and so on.

Knowledge of specific technology details is necessary to perform computer programming. If someone assigns you to write a program in C++, you have to know something about C++ to get your program to work.

You often hear people say that software development knowledge has a 3-year half-life: half of what you need to know today will be obsolete within 3 years. In the domain of technology-related knowledge, that's probably about right. But there is another kind of software development knowledge—a kind that I think of as "software engineering principles"—that does not have a three-year half-life. These software engineering principles are likely to serve a professional programmer throughout his or her career.

McConnell goes on to argue that the body of software engineering knowledge (circa the year 2000) had evolved to a "stable core" that he estimated represented about "75 percent of the knowledge needed to develop a complex system." But what resides within this stable core? As McConnell indicates, core principles—the elemental ideas that guide software engineers in the work that they do—now provide a foundation from which software engineering models, methods, and tools can be applied and evaluated.

1.8 CORE PRINCIPLES

Software engineering is guided by a collection of core principles that help in the application of a meaningful software process and the execution of effective software engineering methods. At the process level, core principles establish a philosophical foundation that guides a software team as it performs framework and umbrella activities, navigates the process flow, and produces a set of software engineering work products.

At the level of practice, core principles establish a collection of values and rules that serve as a guide as you analyze a problem, design a solution, implement and test the solution, and ultimately deploy the software in the user community identified a set of general principles that span software engineering process and practice:

(1) provide value to end users,
(2) keep it simple,
(3) maintain the vision (of the product and the project),
(4) recognize that others consume (and must understand) what you produce,
(5) be open to the future,
(6) plan ahead for reuse, and
(7) think!
Although these general principles are important, they are characterized at such a high level of abstraction that they are sometimes difficult to translate into day-to-day software engineering practice.

1.8.1 Principles that Guide Process

The following set of core principles can be applied to the framework, and by extension, to every software process.

Principle 1: Be agile. Whether the process model you choose is prescriptive or agile.

1. keep your technical approach as simple as possible
2. keep the work products you produce as concise(short) as possible
3. Make decisions locally whenever possible.

Principle 2. Focus on quality at every step. For every process activity, action, and task should focus on the quality of the work product that has been produced.

Principle 3. Be ready to adapt. adapt your approach to conditions imposed by the problem, the people, and the project itself.

Principle 4. Build an effective team. Build a self-organizing team that has mutual trust and respect.

Principle 5. Establish mechanisms for communication and coordination. Projects fail because stakeholders fail to coordinate their efforts to create a successful end product.

Principle 6. Manage change. The methods must be established to manage the way changes are requested, approved, and implemented.

Principle 7. Assess risk. Lots of things can go wrong as software is being developed.

Principle 8. Create work products that provide value for others. Create only those work products that provide value for other process activities, actions and tasks.

1.8.2 Principles That Guide Practice

- Software engineering practice has a single goal i.e.., to deliver on-time, high quality, operational software that contains functions and features that meet the needs of all stakeholders.

- To achieve this goal, should adopt a set of core principles that guide the technical work.

- The following **set of core principles are fundamental to the practice of software engineering**:

Principle 1. Divide and conquer. A large problem is easier to solve if it is subdivided into a collection of elements (or modules or components). Ideally, each element delivers distinct functionality that can be developed.

Principle 2. Understand the use of abstraction. At its core, an abstraction(overview) is a simplification of some complex element of a system used to communicate meaning in a single phrase.

Principle 3. Strive for consistency. Whether it's creating a requirements model, developing a software design, generating source code, or creating test cases. All these are consistent so that the software is easier to develop.

Principle 4. Focus on the transfer of information. Software is about information transfer—from a database to an end user, from an operating system to an application,

Principle 5. Build software that exhibits effective modularity. Modularity provides a mechanism for any complex system can be divided into modules (components).

Principle 6. Look for patterns. Brad Appleton suggests that: The goal of patterns within the software community is to create a body of literature to help software developers resolve recurring problems encountered throughout all of software development process.

Principle 7. When possible, represent the problem and its solution from a number of different perspectives. When a problem and its solution are examined from a number of different perspectives(ways).

Principle 8. Remember that someone will maintain the software. software will be corrected as defects are remove, adapted as its environment changes, and enhanced as stakeholders request more capabilities**.**

1.9 PRINCIPLES THAT GUIDE EACH FRAMEWORK ACTIVITY

1.9.1 Communication Principles

Customer requirements must be gathered through the **communication activity**. Communication has begun.

Principle 1. Listen.

Try to focus on the speaker's words, rather than formulating your response to those words. Ask for clarification if something is unclear, but avoid constant interruptions. Never become contentious in your words or actions (e.g., rolling your eyes or shaking your head) as a person is talking.

Principle 2. Prepare before you communicate. Spend the time to understand the problem before you meet with others. If necessary, do some research to understand business domain jargon. If you have responsibility for conducting a meeting, prepare an agenda in advance of the meeting.

Principle 3. Someone should facilitate the activity. Every communication meeting should have a leader (a facilitator) to keep the conversation moving in a productive direction, (2) to mediate any conflict that does occur, and (3) to ensure than other principles are followed.

Principle 4. Face-to-face communication is best. But it usually works better when some other representation of the relevant information is present. For example, a participant may create a drawing or a "strawman" document that serves as a focus for discussion.

Principle 5. Take notes and document decisions. Things have a way of falling into the cracks. Someone participating in the communication should serve as a "recorder" and write down all important points and decisions.

Principle 6. Strive for collaboration. Collaboration and consensus occur when the collective knowledge of members of the team is used to describe product or system functions or features. Each small collaboration serves to build trust among team members and creates a common goal for the team.

Principle 7. Stay focused; modularize your discussion. The more people involved in any communication, the more likely that discussion will bounce from one topic to the next. The facilitator should keep the conversation modular, leaving one topic only after it has been resolved

Principle 8. If something is unclear, draw a picture. Verbal communication goes only so far. A sketch or drawing can often provide clarity when words fail to do the job.

Principle 9. (a) Once you agree to something, move on. (b) If you can't agree to something, move on. (c) If a feature or function is unclear and cannot be clarified at the moment, move on. Communication, like any software engineering activity, takes time. Rather than iterating endlessly, the people who participate should recognize that many topics require discussion (see Principle 2) and that "moving on" is sometimes the best way to achieve communication agility.

Principle 10. Negotiation is not a contest or a game. It works best when both parties win. There are many instances in which you and other stakeholders must negotiate functions and features, priorities, and delivery dates. If the team has collaborated well, all parties have a common goal. Still, negotiation will demand compromise from all parties.

1.9.2 Planning Principles

- The communication activity helps you to define your overall goals and objectives.
- The planning activity encompasses a set of management and technical practices that enable the software team to define a road map as it travels toward the objectives.

The following principles always apply:

Principle 1. Understand the scope of the project. It's impossible to use a road map if you don't know where you're going. Scope provides the software team with a destination.

Principle 2. Involve stakeholders in the planning activity. Stakeholders define priorities and establish project constraints. To accommodate these realities, software engineers must often negotiate order of delivery, time lines, and other project-related issues.

Principle 3. Recognize that planning is iterative. A project plan is never engraved in stone. As work begins, it is very likely that things will change. As a consequence, the plan must be adjusted to accommodate these changes. In addition, iterative, incremental process models dictate replanning after the delivery of each software increment based on feedback received from users.

Principle 4. Estimate based on what you know. The intent of estimation is to provide an indication of effort, cost, and task duration, based on the team's current understanding of the work to be done. If information is vague or unreliable, estimates will be equally unreliable.

Principle 5. Consider risk as you define the plan. If you have identified risks that have high impact and high probability, contingency planning is necessary. In addition, the project plan (including the schedule) should be adjusted to accommodate the likelihood that one or more of these risks will occur.

Principle 6. Be realistic. People don't work 100 percent of every day. Noise always enters into any human communication. Omissions and ambiguity are facts of life. Change will occur. Even the best software engineers make mistakes. These and other realities should be considered as a project plan is established.

Principle 7. Adjust granularity as you define the plan. Granularity refers to the level of detail that is introduced as a project plan is developed. A "high-granularity" plan provides significant work task detail that is planned over relatively short time increments (so that tracking and control occur frequently). A "low-granularity" plan provides broader work tasks that are planned over longer time periods. In general, granularity moves from high to low as the project time line moves away from the current date. Over the next few weeks or months, the project can be planned in significant detail. Activities that won't occur for many months do not require high granularity (too much can change).

Principle 8. Define how you intend to ensure quality. The plan should identify how the software team intends to ensure quality. If technical reviews3 are to be conducted, they should be scheduled. If pair programming (Chapter 3) is to be used during construction, it should be explicitly defined within the plan.

Principle 9. Describe how you intend to accommodate change. Even the best planning can be obviated by uncontrolled change. You should identify how changes are to be accommodated as software engineering work proceeds. For example, can the customer request a change at any time? If a change is requested, is the team obliged to implement it immediately? How is the impact and cost of the change assessed?

Principle 10. Track the plan frequently and make adjustments as required. Software projects fall behind schedule one day at a time. Therefore, it makes sense to track progress on a daily basis, looking for problem areas and situations in which scheduled work does not conform to actual work conducted. When slippage is encountered, the plan is adjusted accordingly.

1.9.3 Modeling Principles

Create models to gain a better understanding of the actual entity to be built. The modeling principles are:

Principle 1. The primary goal of the software team is to build software, not create models. Agility means getting software to the customer in the fastest possible time. Models that make this

happen are worth creating, but models that slow the process down or provide little new insight should be avoided.

Principle 2. Travel light—don't create more models than you need. Every model that is created must be kept up-to-date as changes occur. More importantly, every new model takes time that might otherwise be spent on construction (coding and testing). Therefore, create only those models that make it easier and faster to construct the software.

Principle 3. Strive to produce the simplest model that will describe the problem or the software. Don't overbuild the software [Amb02b]. By keeping models simple, the resultant software will also be simple. The result is software that is easier to integrate, easier to test, and easier to maintain (to change). In addition, simple models are easier for members of the software team to understand and critique, resulting in an ongoing form of feedback that optimizes the end result.

Principle 4. Build models in a way that makes them amenable to change. Assume that your models will change, but in making this assumption don't get sloppy. For example, since requirements will change, there is a tendency to give requirements models short shrift. Why? Because you know that they'll change anyway. The problem with this attitude is that without a reasonably complete requirements model, you'll create a design (design model) that will invariably miss important functions and features.

Principle 5. Be able to state an explicit purpose for each model that is created. Every time you create a model, ask yourself why you're doing so. If you can't provide solid justification for the existence of the model, don't spend time on it.

Principle 6. Adapt the models you develop to the system at hand. It may be necessary to adapt model notation or rules to the application; for example, a video game application might require a different modeling technique than real-time, embedded software that controls an automobile engine.

Principle 7. Try to build useful models, but forget about building perfect models. When building requirements and design models, a software engineer reaches a point of diminishing returns. That is, the effort required to make the model absolutely complete and internally consistent is not worth the benefits of these properties. Am I suggesting that modeling should be sloppy or low quality? The answer is "no." But modeling should be conducted with an eye to the next software engineering steps. Iterating endlessly to make a model "perfect" does not serve the need for agility.

Principle 8. Don't become dogmatic about the syntax of the model. If it communicates content successfully, representation is secondary Although everyone on a software team should try to use consistent notation during modeling, the most important characteristic of the model is to communicate information that enables the next software engineering task. If a model does this successfully, incorrect syntax can be forgiven.

Principle 9. If your instincts tell you a model isn't right even though it seems okay on paper, you probably have reason to be concerned. If you are an experienced software engineer, trust your instincts. Software work teaches many lessons—some of them on a subconscious level. If something tells you that a design model is doomed to fail (even though you can't prove it explicitly), you have reason to spend additional time examining the model or developing a different one.

Principle 10. Get feedback as soon as you can. Every model should be reviewed by members of the software team. The intent of these reviews is to provide feedback that can be used to correct modeling mistakes, change misinterpretations, and add features or functions that were inadvertently omitted.

Requirements modeling principles.

Principle 1. The information domain of a problem must be represented and understood. The information domain encompasses the data that flow into the system (from end users, other systems, or external devices), the data that flow out of the system (via the user interface, network interfaces, reports, graphics, and other means), and the data stores that collect and organize persistent data objects (i.e., data that are maintained permanently).

Principle 2. The functions that the software performs must be defined. Software functions provide direct benefit to end users and also provide internal support for those features that are user visible. Some functions transform data that flow into the system. In other cases, functions effect some level of control over internal software processing or external system elements. Functions can be described at many different levels of abstraction, ranging from a general statement of purpose to a detailed description of the processing elements that must be invoked.

Principle 3. The behavior of the software (as a consequence of external events) must be represented. The behavior of computer software is driven by its interaction with the external environment. Input provided by end users, control data provided by an external system, or monitoring data collected over a network all cause the software to behave in a specific way.

Principle 4. The models that depict information, function, and behavior must be partitioned in a manner that uncovers detail in a layered (or hierarchical) fashion. Requirements modeling is the first step in software engineering problem solving. It allows you to better understand the problem and establishes a basis for the solution (design). Complex problems are difficult to solve in their entirety. For this reason, you should use a divide-and-conquer strategy. A large, complex problem is divided into subproblems until each subproblem is relatively easy to understand. This concept is called partitioning or separation of concerns, and it is a key strategy in requirements modeling.

Principle 5. The analysis task should move from essential information toward implementation detail. Requirements modeling begins by describing the problem from the end-user's perspective. The "essence" of the problem is described without any consideration of how a solution will be

implemented. For example, a video game requires that the player "instruct" its protagonist on what direction to proceed as she moves into a dangerous maze. That is the essence of the problem. Implementation detail (normally described as part of the design model) indicates how the essence will be implemented. For the video game, voice input might be used. Alternatively,

Design Modeling Principles

Principle 1. Design should be traceable to the requirements model. The requirements model describes the information domain of the problem, user-visible functions, system behavior, and a set of requirements classes that package business objects with the methods that service them. The design model translates this information into an architecture, a set of subsystems that implement major functions, and a set of components that are the realization of requirements classes. The elements of the design model should be traceable to the requirements model.

Principle 2. Always consider the architecture of the system to be built. Software architecture is the skeleton of the system to be built. It affects interfaces, data structures, program control flow and behavior, the manner in which testing can be conducted, the maintainability of the resultant system, and much more. For all of these reasons, design should start with architectural considerations. Only after the architecture has been established should component-level issues be considered.

Principle 3. Design of data is as important as design of processing functions. Data design is an essential element of architectural design. The manner in which data objects are realized within the design cannot be left to chance. A well-structured data design helps to simplify program flow, makes the design and implementation of software components easier, and makes overall processing more efficient.

Principle 4. Interfaces (both internal and external) must be designed with care. The manner in which data flows between the components of a system has much to do with processing efficiency, error propagation, and design simplicity. A well-designed interface makes integration easier and assists the tester in validating component functions.

Principle 5. User interface design should be tuned to the needs of the end user. However, in every case, it should stress ease of use. The user interface is the visible manifestation of the software. No matter how sophisticated its internal functions, no matter how comprehensive its data structures, no matter how well designed its architecture, a poor interface design often leads to the perception that the software is "bad."

Principle 6. Component-level design should be functionally independent. Functional independence is a measure of the "single-mindedness" of a software component. The functionality that is delivered by a component should be cohesive—that is, it should focus on one and only one function or subfunction.

Principle 7. Components should be loosely coupled to one another and to the external environment. Coupling is achieved in many ways— via a component interface, by messaging, through global data. As the level of coupling increases, the likelihood of error propagation also increases and the overall maintainability of the software decreases. Therefore, component coupling should be kept as low as is reasonable.

Principle 8. Design representations (models) should be easily understandable. The purpose of design is to communicate information to practitioners who will generate code, to those who will test the software, and to others who may maintain the software in the future. If the design is difficult to understand, it will not serve as an effective communication medium.

Principle 9. The design should be developed iteratively. With each iteration, the designer should strive for greater simplicity. Like almost all creative activities, design occurs iteratively. The first iterations work to refine the design and correct errors,

1.9.4 Construction Principles

The construction activity encompasses a set of coding and testing tasks that lead to operational software that is ready for delivery to the customer or end user. coding may be

(1) the direct creation of programming language source code (e.g., Java),
(2) the automatic generation of source code using an Intermediate design like representation of the component to be built.

The initial focus of testing is at the component level, often called **unit testing.**

Other levels of testing include
i) **integration testing** (conducted as the system is constructed).
ii) **validation testing** that assesses whether requirements have been met for the complete system (or software increment)
(iii) **acceptance testing** that is conducted by the customer in an effort to exercise all required features and functions.

The following set of fundamental principles and concepts are applicable to coding and testing:

Coding Principles. The principles that guide the coding task are closely aligned with programming style, programming languages, and programming methods. However, there are a number of fundamental principles that can be stated:

Preparation principles: Before you write one line of code, be sure you

- Understand of the problem you're trying to solve.
- Understand basic design principles and concepts.
- Pick a programming language that meets the needs of the software to be built and the environment in which it will operate.
- Select a programming environment that provides tools that will make your work easier.
- Create a set of unit tests that will be applied once the component you code is completed.

Programming principles: As you begin writing code, be sure you

- Constrain your algorithms by following structured programming practice.
- Consider the use of pair programming.
- Select data structures that will meet the needs of the design.
- Understand the software architecture and create interfaces that are consistent with it.
- Keep conditional logic as simple as possible.
- Create nested loops in a way that makes them easily testable.
- Select meaningful variable names and follow other local coding standards.
- Write code that is self-documenting.
- Create a visual layout (e.g., indentation and blank lines) that aids understanding.

Validation Principles: After you've completed your first coding pass, be sure you

- Conduct a code walkthrough when appropriate.
- Perform unit tests and correct errors you've uncovered.
- Refactor the code.

Testing Principles: Glen Myers states a number of rules that can serve well as testing objectives:

- Testing is a process of executing a program with the intent of finding an error.
- A good test case is one that has a high probability of finding an as-yet undiscovered error.
- A successful test is one that uncovers an as-yet-undiscovered error.

Davis suggests a set of testing principles that have been adapted for use.

Principle 1. All tests should be traceable to customer requirements. The objective of software testing is to uncover errors. It follows that the most severe defects (from the customer's point of view) are those that cause the program to fail to meet its requirements.

Principle 2. Tests should be planned long before testing begins. Test planning can begin as soon as the requirements model is complete. Detailed definition of test cases can begin as soon as the design model has been solidified. Therefore, all tests can be planned and designed before any code has been generated.

Principle 3. The Pareto principle applies to software testing. In this context the Pareto principle implies that 80 percent of all errors uncovered during testing will likely be traceable to 20 percent of all program components. The problem, of course, is to isolate these suspect components and to thoroughly test them.

Principle 4. Testing should begin "in the small" and progress toward testing "in the large." The first tests planned and executed generally focus on individual components. As testing progresses, focus shifts in an attempt to find errors in integrated clusters of components and ultimately in the entire system.

Principle 5. Exhaustive testing is not possible. The number of path permutations for even a moderately sized program is exceptionally large. For this reason, it is impossible to execute every combination of paths during testing. It is possible, however, to adequately cover program logic and to ensure that all conditions in the component-level design have been exercised.

Deployment Principles

Principle 1. Customer expectations for the software must be managed. Too often, the customer expects more than the team has promised to deliver, and disappointment occurs immediately. This results in feedback that is not productive and ruins team morale. In her book on managing expectations, Naomi Karten states: "The starting point for managing expectations is to become more conscientious about what you communicate and how." She suggests that a software engineer must be careful about sending the customer conflicting messages (e.g., promising more than you can reasonably deliver in the time frame provided or delivering more than you promise for one software increment and then less than promised for the next).

Principle 2. A complete delivery package should be assembled and tested. A CD-ROM or other media (including Web-based downloads) containing all executable software, support data files, support documents, and other relevant information should be assembled and thoroughly beta-tested with actual users. All installation scripts and other operational features should be thoroughly exercised in as many different computing configurations (i.e., hardware, operating systems, peripheral devices, networking arrangements) as possible.

Principle 3. A support regime must be established before the software is delivered. An end user expects responsiveness and accurate information when a question or problem arises. If support is ad hoc, or worse, nonexistent, the customer will become dissatisfied immediately. Support should be planned, support materials should be prepared, and appropriate recordkeeping mechanisms should be established so that the software team can conduct a categorical assessment of the kinds of support requested.

Principle 4. Appropriate instructional materials must be provided to end users. The software team delivers more than the software itself. Appropriate training aids (if required) should be developed; troubleshooting guidelines should be provided, and when necessary, a "what's different about this software increment" description should be published.

Principle 5. Buggy software should be fixed first, delivered later. Under time pressure, some software organizations deliver low-quality increments with a warning to the customer those bugs "will be fixed in the next release." This is a mistake. There's a saying in the software business: "Customers will forget you delivered a high-quality product a few days late, but they will never forget the problems that a low-quality product caused them. The software reminds them every day.".