



AGILE DEVELOPMENT

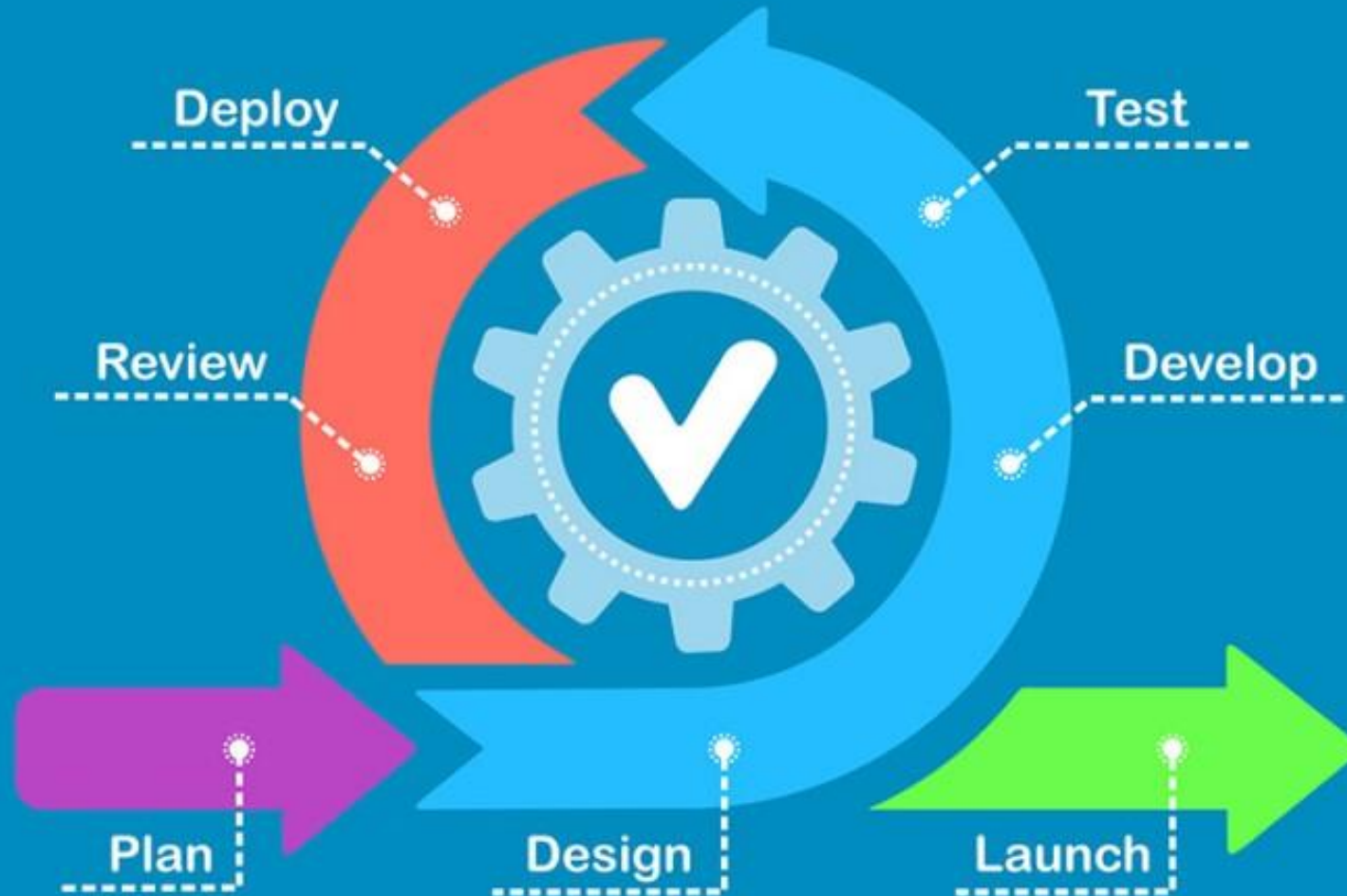
BY: STUTI SMART

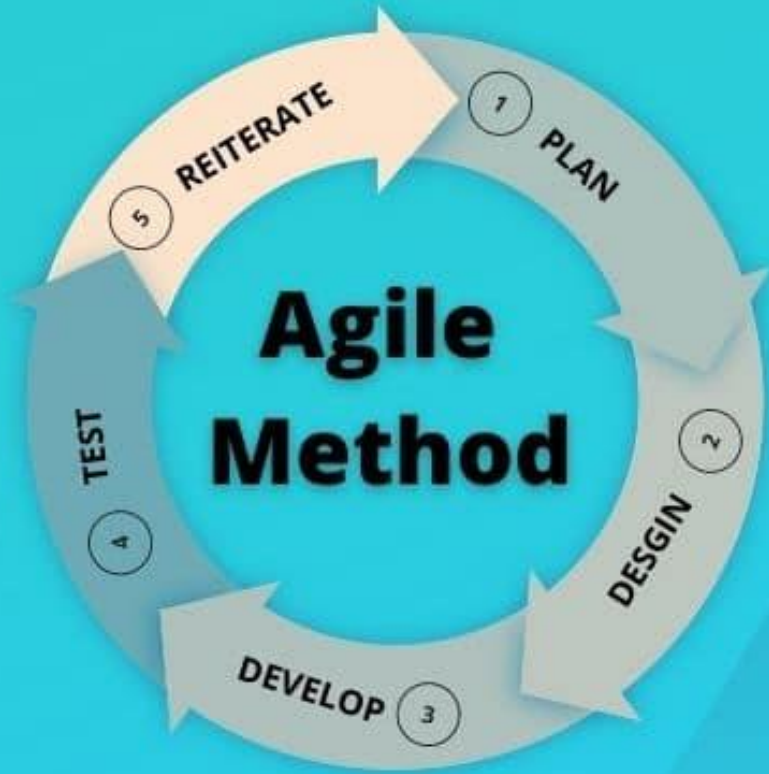
Chapter 1

What Is Agility?

- ▶ Agile software development is a way of creating software that values collaboration, feedback, and adaptation over rigid plans and documentation. Agile software development teams work in short cycles called sprints, where they deliver working software to the customer frequently and respond to changing requirements quickly. Agile software development is based on the Agile Manifesto, which outlines four core values and twelve principles for agile teams. Some of the most popular agile methods are Scrum, Kanban, and Extreme Programming (XP). Agile software development is better than traditional methods because it allows teams to deliver value faster, improve quality, reduce risk, increase customer satisfaction, and embrace change.

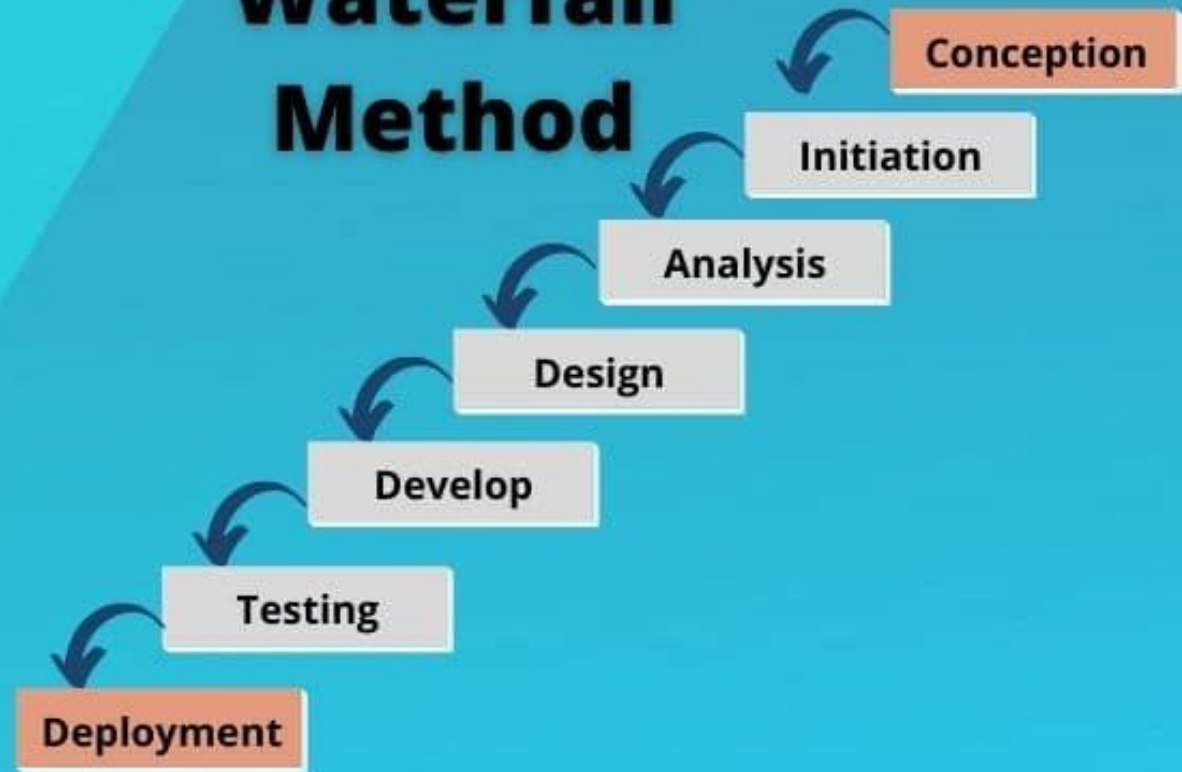
AGILE





- Continuous cycles
- Small, high-functioning, collaborative teams
- flexible/continuous evolution
- Customer involvement

Waterfall Method

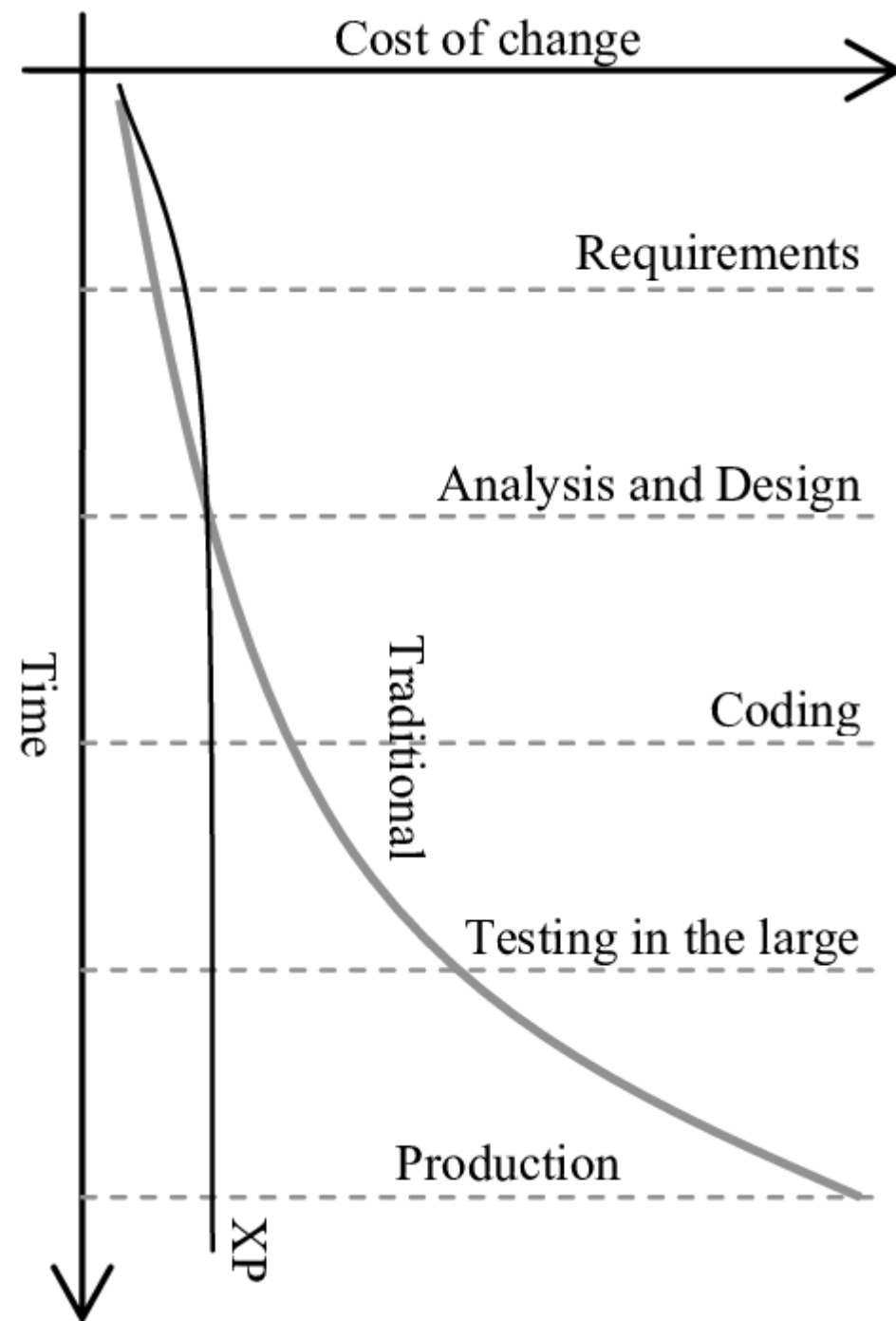


- Sequential/linear stages
- Upfront planning and in-depth documentation
- Best for simple, unchanging projects
- Close project manager involvement

Agility And Cost of Change.

- ▶ One of the main benefits of agility is the ability to respond quickly and effectively to changing customer needs, market conditions, and business opportunities. However, agility also comes with a cost: the cost of change. The cost of change refers to the effort, time, and resources required to implement a change in a software project, such as adding a new feature, fixing a bug, or improving the quality. The cost of change depends on many factors, such as the size and complexity of the project, the quality of the code and design, the level of automation and testing, and the skills and experience of the team. The cost of change can also vary depending on when the change is introduced: changes made early in the project tend to have lower costs than changes made later in the project.

- ▶ Therefore, agility requires a careful balance between delivering value to customers and managing the cost of change. To achieve this balance, agile teams need to adopt practices and principles that help them reduce the cost of change and increase their ability to adapt. Some of these practices and principles include:
 - ▶ **Iterative and incremental development:** delivering working software in small batches that provide value to customers and feedback to the team.
 - ▶ **Test-driven development:** writing automated tests before writing code to ensure that the code meets the requirements and prevents defects.
 - ▶ **Continuous integration and delivery:** integrating and deploying code frequently to ensure that the software is always in a releasable state.
 - ▶ **Refactoring:** improving the structure and design of the code without changing its behavior to make it easier to understand, modify, and maintain.
 - ▶ **Simple design:** creating software that is easy to change by avoiding unnecessary complexity, duplication, and dependencies.
 - ▶ **Collective ownership:** sharing responsibility for the quality and evolution of the code among all team members.
 - ▶ **Self-organizing teams:** empowering teams to make decisions about how they work and what they deliver based on their knowledge and feedback.
- ▶ By following these practices and principles, agile teams can achieve a high level of agility while minimizing the cost of change. This way, they can deliver software that meets customer needs and expectations while maintaining quality and efficiency.



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. 68 PART ONE THE SOFTWARE PROCESS Cost of change using conventional software processes Cost of change using agile processes Idealized cost of change using agile process Development schedule progress Development cost Change costs as a function of time in development quote: “Agility is dynamic, content specific, aggressively change embracing, and growth oriented.” –Steven Goldman et al. An agile process reduces the cost of change because software is released in increments and change can be better controlled within an increment.
 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 (from a planning point of view) as we might like.

Agility Principles

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

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity—the art of maximizing the amount of work not done—is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

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

Human Factors

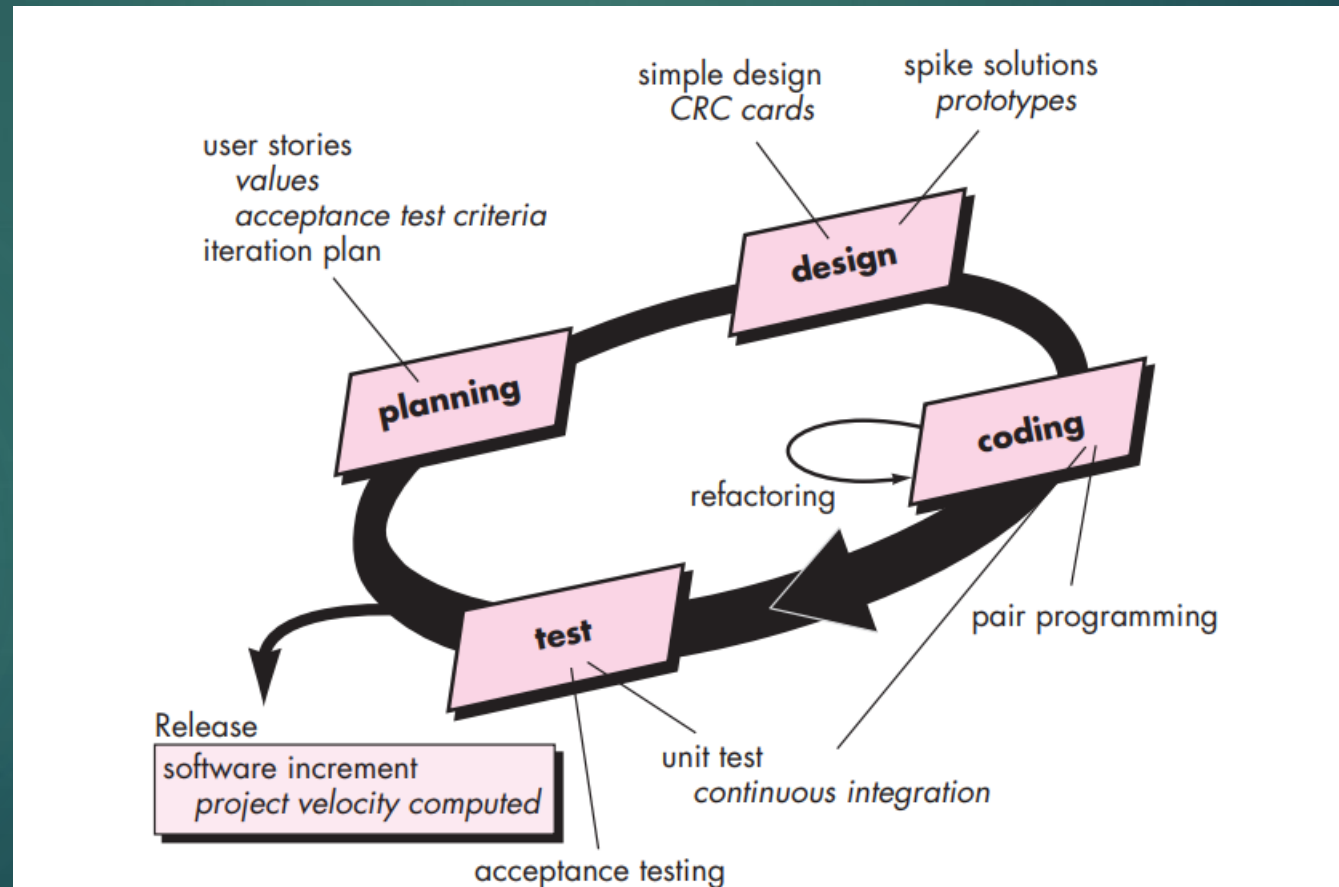
- ▶ Proponents of agile software development take great pains to emphasize the importance of “people factors.” As Cockburn and Highsmith state, “Agile development focuses on the talents and skills of individuals, molding the process to specific people and teams.” The key point in this statement is that the process molds to the needs of the people and team, not the other way around. If members of the software team are to drive the characteristics of the process that is applied to build software, a number of key traits must exist among the people on an agile team and the team itself.
 1. **Competence.** In an agile development (as well as software engineering) 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

1. **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.
2. **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.
3. **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. In some cases, the team must accept the fact that the problem they are solving today may not be the problem that needs to be solved tomorrow. However, lessons learned from any problem-solving activity (including those that solve the wrong problem) may be of benefit to the team later in the project.
4. **Mutual trust and respect:** The agile team must become what DeMarco and Lister [DeM98] call a “jelled” team (Chapter 24). 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.”
5. **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. In essence, the team serves as its own management. K


Extreme Programming (XP)

- ▶ In order to illustrate an agile process in a bit more detail, I'll provide you with an overview of Extreme Programming (XP), the most widely used approach to agile software development. Although early work on the ideas and methods associated with XP occurred during the late 1980s, the seminal work on the subject has been written by Kent Beck. More recently, a variant of XP, called Industrial XP (IXP) has been proposed. IXP refines XP and targets the agile process specifically for use within large organizations.
 - ▶ **XP Values:** Beck defines a set of five values that establish a foundation for all work performed as part of XP—communication, simplicity, feedback, courage, and respect. Each of these values is used as a driver for specific XP activities, actions, and tasks. In order to achieve effective communication between software engineers and other stakeholders (e.g., to establish required features and functions for the software), XP emphasizes close, yet informal (verbal) collaboration between customers and developers, the establishment of effective metaphors³ for communicating important concepts, continuous feedback, and the avoidance of voluminous documentation as a communication medium
 - ▶ **The XP Process:** Extreme Programming uses an object-oriented approach (Appendix 2) 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. The XP process notes some of the key ideas and tasks that are associated with each framework activity. Key XP activities are summarized in the paragraphs that follow

- **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 describe required output, features, and functionality for software to be built.




- ▶ **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 (because the developer assumes it will be required later) is discouraged.⁶ XP encourages the use of CRC cards as an effective mechanism for thinking about the software in an object-oriented context. CRC (class-responsibilitycollaborator) cards identify and organize the object-oriented classes⁷ that are relevant to the current software increment. The XP team conducts the design exercise using a process similar to the one described in Chapter 8. The CRC cards are the only design work product produced as part of the XP process.
- ▶ **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).⁸ Once the unit test⁹ has been created, the developer is better able to focus on what must be implemented to pass the test. Nothing extraneous is added. 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 (and one of the most talked about aspects of XP) 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 realtime 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 (a required part of XP) 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 (Chapter 17) that helps to uncover errors early.

- 
- ▶ **Testing:** I have already noted that 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 (hence, they can be executed easily and repeatedly). This encourages a regression testing strategy (Chapter 17) whenever code is modified (which is often, given the XP refactoring philosophy). As the individual unit tests are organized into a “universal testing suite” [Wel99], 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 [Wel99] states: “Fixing small problems every few hours takes less time than fixing huge problems just before the deadline.” XP acceptance tests, also called customer tests, are specified by the customer and focus on overall system features and functionality that are visible and reviewable by the customer. Acceptance tests are derived from user stories that have been implemented as part of a software release.

Industrial XP

- ▶ Joshua Kerievsky [Ker05] 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.
 - ▶ Project community.
 - ▶ Project chartering.
 - ▶ Test-driven management.
 - ▶ Retrospectives.
 - ▶ Continuous learning.

- 
- ▶ **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” [Ker05]. 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” [Ker05] and then defines mechanisms for determining whether or not these destinations have been reached.
- ▶ **Retrospectives:** An IXP team conducts a specialized technical review (Chapter 15) after a software increment is delivered. Called a retrospective, the review examines “issues, events, and lessons-learned” [Ker05] 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


The XP Debate

- ▶ All new process models and methods spur worthwhile discussion and in some instances heated debate. Extreme Programming has done both. In an interesting book that examines the efficacy of XP, Stephens and Rosenberg [Ste03] argue that many XP practices are worthwhile, but others have been overhyped, and a few are problematic. The authors suggest that the codependent nature of XP practices are both its strength and its weakness. Because many organizations adopt only a subset of XP practices, they weaken the efficacy of the entire process. Proponents counter that XP is continuously evolving and that many of the issues raised by critics have been addressed as XP practice matures. Among the issues that continue to trouble some critics of XP are:

- ▶ **Requirements volatility:** Because 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. Proponents argue that this happens regardless of the process that is applied and that XP provides mechanisms for controlling scope creep.
- ▶ **Conflicting customer needs:** Many projects have multiple customers, each with his own set of needs. In XP, the team itself is tasked with assimilating the needs of different customers, a job that may be beyond their scope of authority.
- ▶ **Requirements are expressed informally:** User stories and acceptance tests are the only explicit manifestation of requirements in XP. Critics argue that a more formal model or specification is often needed to ensure that omissions, inconsistencies, and errors are uncovered before the system is built. Proponents counter that the changing nature of requirements makes such models and specification obsolete almost as soon as they are developed.
- ▶ **Lack of formal design:** XP deemphasizes the need for architectural design and in many instances, suggests that design of all kinds should be relatively informal. Critics argue that when complex systems are built, design must be emphasized to ensure that the overall structure of the software will exhibit quality and maintainability. XP proponents suggest that the incremental nature of the XP process limits complexity (simplicity is a core value) and therefore reduces the need for extensive design.
- ▶ You should note that every software process has flaws and that many software organizations have used XP successfully. The key is to recognize where a process may have weaknesses and to adapt it to the specific needs of your organization.

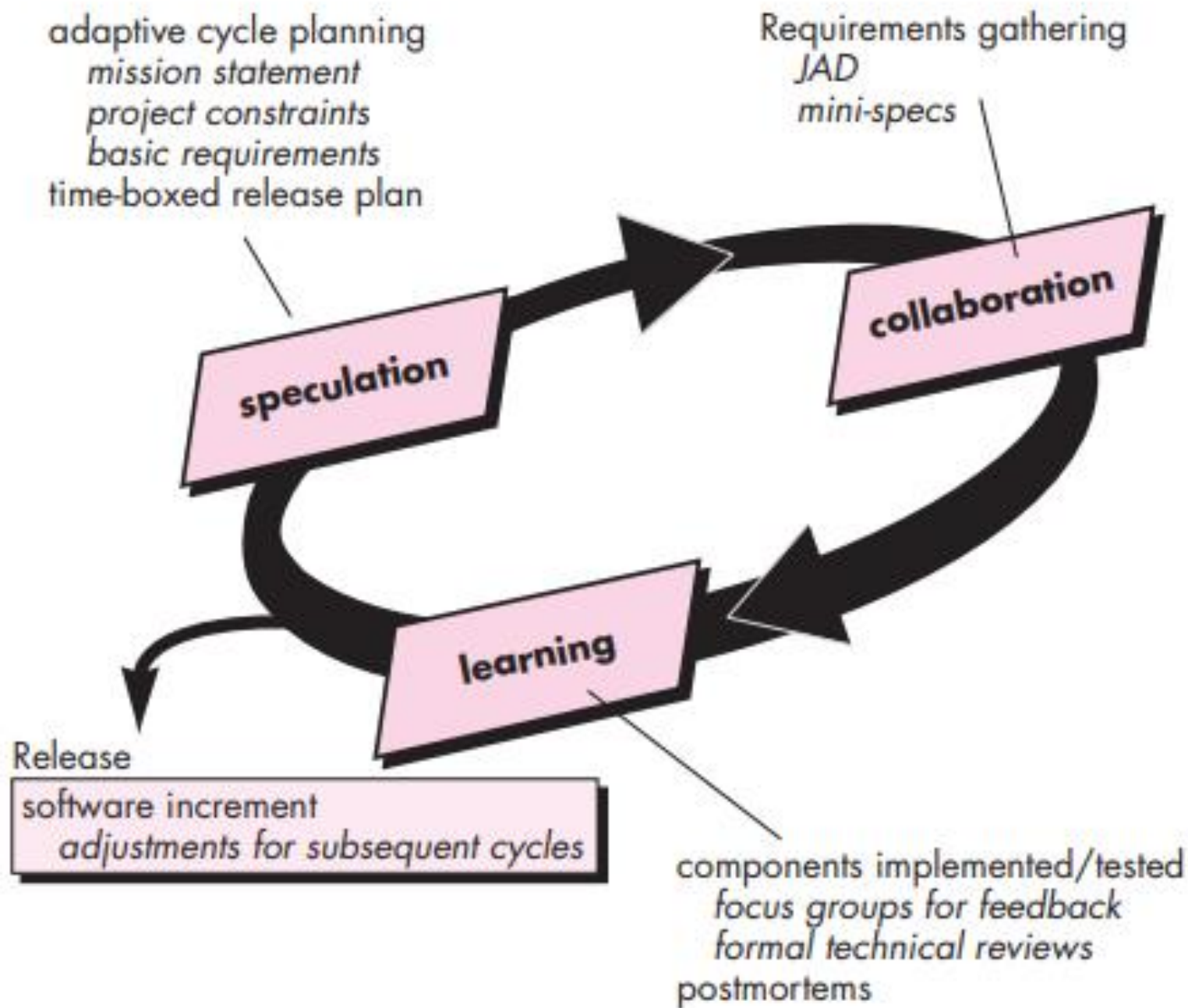
Other Agile process Models

- ▶ The history of software engineering is littered with dozens of obsolete process descriptions and methodologies, modeling methods and notations, tools, and technology. Each flared in notoriety and was then eclipsed by something new and (purportedly) better. With the introduction of a wide array of agile process models—each contending for acceptance within the software development community—the agile movement is following the same historical path.¹¹ As I noted in the last section, the most widely used of all agile process models is Extreme Programming (XP). But many 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)
 - ▶ In the sections that follow, I present a very brief overview of each of these agile process models. It is important to note that all agile process models conform (to a greater or lesser degree) to the Manifesto for Agile Software Development and the principles noted. For additional detail, refer to the references noted in each subsection or for a survey, examine the “agile software development” entry in Wikipedia.

Adaptive Software Development (ASD)

- ▶ Adaptive Software Development (ASD) has been proposed by Jim Highsmith [Hig00] as a technique for building complex software and systems. The philosophical underpinnings of ASD focus on human collaboration and team self-organization.
- ▶ Highsmith argues that an agile, adaptive development approach based on collaboration is “as much a source of order in our complex interactions as discipline and engineering.” He defines an ASD “life cycle” (Figure 3.3) that incorporates three phases, speculation, collaboration, and learning



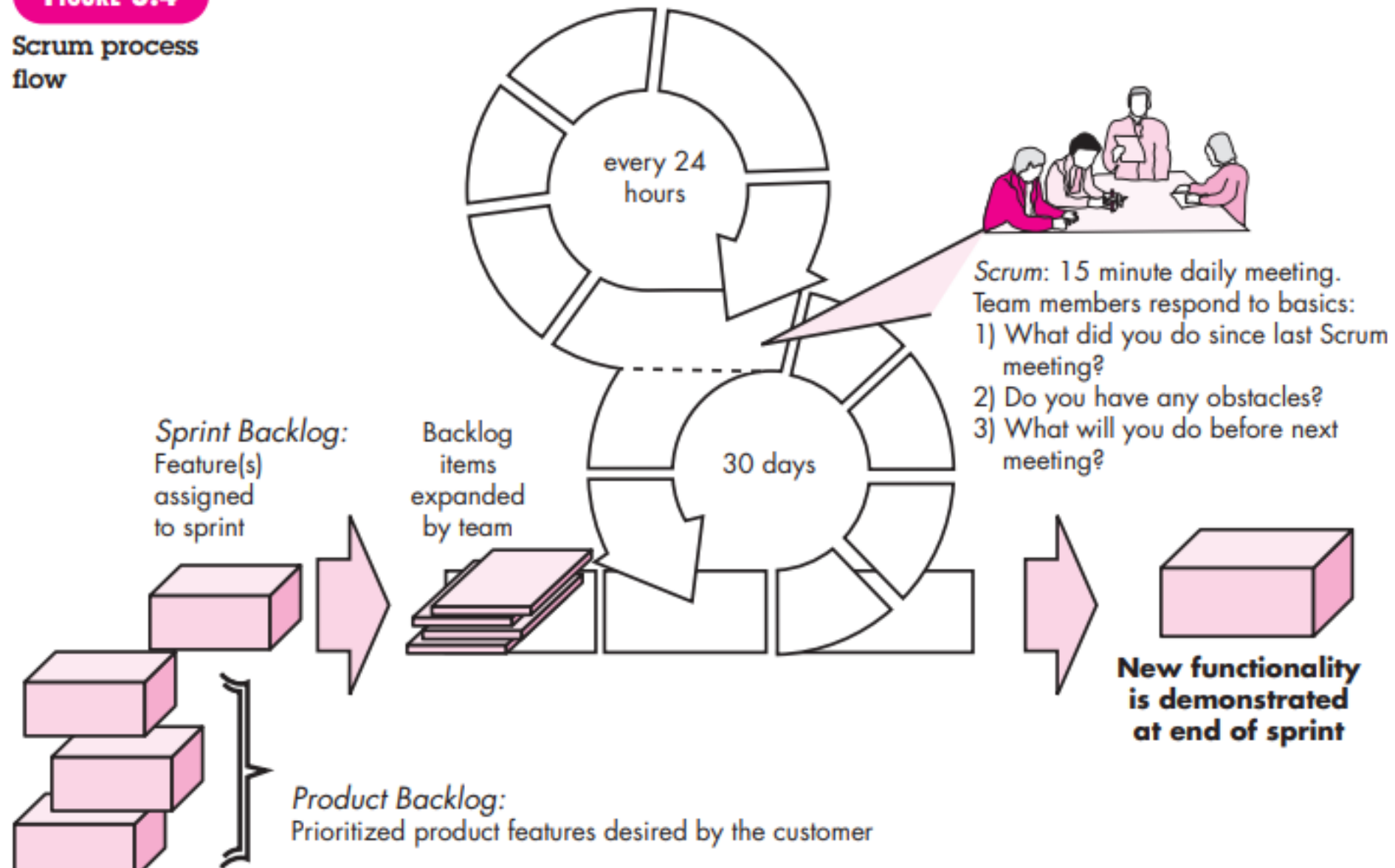
- ▶ As members of an ASD team begin to develop the components that are part of an adaptive cycle, the emphasis is on “learning” as much as it is on progress toward a completed cycle. In fact, Highsmith [Hig00] argues that software developers often overestimate their own understanding (of the technology, the process, and the project) and that learning will help them to improve their level of real understanding. ASD teams learn in three ways: focus groups (Chapter 5), technical reviews (Chapter 14), and project postmortems.
- ▶ The ASD philosophy has merit regardless of the process model that is used. 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 (the name is derived from an activity that occurs during a rugby match¹³) is an agile software development method that was conceived by Jeff Sutherland and his development team in the early 1990s. In recent years, further development on the Scrum methods has been performed by Schwaber and Beedle.
- ▶ 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 (discussed in the following paragraph) called a sprint. The work conducted within a sprint (the number of sprints required for each framework activity will vary depending on product complexity and size) is adapted to the problem at hand and is defined and often modified in real time by the Scrum team.
- ▶ Scrum emphasizes the use of a set of software process patterns [Noy02] 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 (this is how changes are introduced). 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-box¹⁴ (typically 30 days)

FIGURE 3.4

Scrum process
flow



► 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 [Noy02]:

- 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” [Bee99] and thereby promote a self-organizing team structure. Demos—deliver the software increment to the customer so that functionality that has been implemented can be demonstrated and evaluated by the customer. 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. Beedle and his colleagues [Bee99] present a comprehensive discussion of these patterns in which they state: “Scrum assumes up-front the existence of chaos. . . .” The Scrum process patterns enable a software team to work successfully in a world where the elimination of uncertainty is impossible

Dynamic Systems Development Method (DSDM)

- ▶ The Dynamic Systems Development Method (DSDM) [Sta97] 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” [CCS02]. 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 Consortium (www.dsdm.org) is a worldwide group of member companies that collectively take on the role of “keeper” of the method. The consortium has defined an agile process model, called the DSDM life cycle that defines three different iterative cycles, preceded by two additional life cycle activities:

- ▶ Feasibility study—establishes the basic business requirements and constraints associated with the application to be built and then assesses whether the application is a viable candidate for the DSDM process.
- ▶ Business study—establishes the functional and information requirements that will allow the application to provide business value; 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. (Note: All DSDM prototypes are intended to evolve into the deliverable application.) 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 (an “operationalized” prototype) 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 [Coc05] and Jim Highsmith [Hig02b] created the Crystal family of agile methods¹⁵ in order to achieve a software development approach that puts a premium on “maneuverability” during what Cockburn characterizes as “a resourcelimited, 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”
- ▶ To achieve maneuverability, Cockburn and Highsmith have defined a set of methodologies, each with core elements that are common to all, and roles, process patterns, work products, and practice that are unique to each. 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.

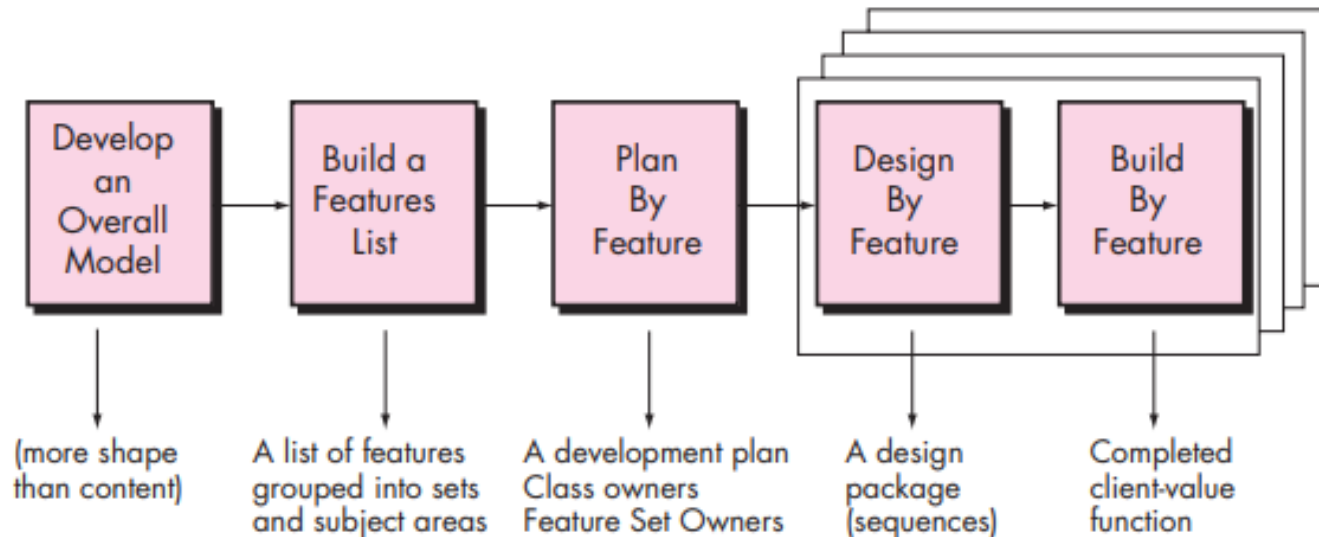
Feature Driven Development (FDD)


- ▶ Feature Driven Development (FDD) was originally conceived by Peter Coad and his colleagues [Coa99] as a practical process model for object-oriented software engineering. Stephen Palmer and John Felsing [Pal02] 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 (Chapter 16), 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” [Coa99]. 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.

FIGURE 3.5

Feature Driven Development [Coa99] (with permission)



- 
- ▶ For example: Making a product sale is a feature set that would encompass the features noted earlier and others.
 - ▶ The FDD approach defines five “collaborating” [Coa99] framework activities (in FDD these are called “processes”)
 - ▶ FDD provides greater emphasis on project management guidelines and techniques than many other agile methods. As projects grow in size and complexity, ad hoc project management is often inadequate. It is essential for developers, their managers, and other stakeholders to understand project status—what accomplishments have been made and problems have been encountered. If deadline pressure is significant, it is critical to determine if software increments (features) are properly scheduled. To accomplish this, FDD defines six milestones during the design and implementation of a feature: “design walkthrough, design, design inspection, code, code inspection, promote to build” [Coa99].

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 ([Pop03], [Pop06a]) as eliminate waste, build quality in, create knowledge, defer commitment, deliver fast, respect people, and optimize the whole.
- ▶ Each of these principles can be adapted to the software process. For example, eliminate waste within the context of an agile software project can be interpreted to mean [Das05]:
 - ▶ adding no extraneous features or functions,
 - ▶ assessing the cost and schedule impact of any newly requested requirement,
 - ▶ removing any superfluous process steps,
 - ▶ establishing mechanisms to improve the way team members find information,
 - ▶ ensuring the testing finds as many errors as possible,
 - ▶ reducing the time required to request and get a decision that affects the software or the process that is applied to create it, and
 - ▶ streamlining the manner in which information is transmitted to all stakeholders involved in the process. For a detailed discussion of LSD and pragmatic guidelines for implementing the process, you should examine [Pop06a] and [Pop06b].

Agile Modeling (AM)

- ▶ There are many situations in which software engineers must build large, businesscritical systems. The scope and complexity of such systems must be modeled so that
 - ▶ all constituencies can better understand what needs to be accomplished,
 - ▶ the problem can be partitioned effectively among the people who must solve it, and
 - ▶ quality can be assessed as the system is being engineered and built.
- ▶ Although AM suggests a wide array of “core” and “supplementary” modeling principles, those that make AM unique are [Amb02a]:
 - ▶ **Model with a purpose**
 - ▶ A developer who uses AM should have a specific goal (e.g., to communicate information to the customer or to help better understand some aspect of the software) 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 [Amb02a] 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 (hence potentially enhancing communication within your team as well as with project stakeholders).”

▶ **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.**

- ▶ 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. A major segment of the software engineering community has adopted the Unified Modeling Language (UML) 16 as the preferred method for representing analysis and design models. The Unified Process (Chapter 2) has been developed to provide a framework for the application of UML. Scott Ambler [Amb06] has developed a simplified version of the UP that integrates his agile modeling philosophy

Agile Unified Process (AUP)

- ▶ The Agile Unified Process (AUP) adopts a “serial in the large” and “iterative in the small” [Amb06] philosophy for building computer-based systems. By adopting the classic UP phased activities—inception, elaboration, construction, and transition—AUP provides a serial overlay (i.e., a linear sequence of software engineering activities) 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 [Amb06]:
 - ▶ **Modeling.** UML representations of the business and problem domains are created. However, to stay agile, these models should be “just barely good enough” [Amb06] to allow the team to proceed.
 - ▶ **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 discussed in Chapters 1 and 2, 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 (Chapter 22) addresses change management, risk management, and the control of any persistent work products¹⁷ that are produced by the team. Project management tracks and controls the progress of the team and coordinates team activities.
 - ▶ **Environment management.** Environment management coordinates a process infrastructure that includes standards, tools, and other support technology available to the team.

A Tool Set For The Agile Process

- ▶ 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].
- ▶ Are any of these things really tools? They are, if they facilitate the work performed by an agile team member and enhance the quality of the end product.

**THANK
YOU**

