A Survey of Agile Development Methodologies

Agile development methodologies are emerging in the software industry. In this chapter, we provide an introduction to agile development methodologies and an overview of four specific methodologies:

- Extreme Programming
- Crystal Methods
- Scrum
- Feature Driven Development

Plan-driven methods work best when developers can determine the requirements in advance . . . and when the requirements remain relatively stable, with change rates on the order of one percent per month.
-- Barry Boehm [11]

Plan-driven methods are those that begin with the solicitation and documentation of a set of requirements that is as complete as possible. Based on these requirements, one can then formulate a plan of development. Usually, the more complete the requirements, the better the plan. Some examples of plan-driven methods are various waterfall approaches and others such as the Personal Software Process (PSP) [28] and the Rational Unified Process (RUP) [30, 31]. An underlying assumption in plan-driven processes is that the requirements are relatively static. On the other hand, iterative methods, such as spiral-model based approaches [12, 14], evolutionary processes described in [5, 22, 32, 33], and recently agile approaches [45] count on change and recognize that the only constant is change. The question is only of the degree and the impact of the change. Beginning in the mid-1990's, practitioners began finding the rate of change in software requirements increasing well beyond the capabilities of classical development methodologies [11, 27]. The software industry, software technology, and customers expectations were moving very quickly and the customers were becoming increasingly less able to fully state their needs up front. As a result, agile methodologies and practices emerged as an explicit attempt to more formally embrace higher rates of requirements change. .

In this chapter, we provide background information on agile principles, and we provide an overview of three agile methodologies. For each of these methodologies, we will present an overview, the main roles involved in the methodology, the documents and artifacts produced, and the development process.

1 What is Agility in Software Development?

In this section, we discuss the model underlying agile software development.

1.1 Agile Model

Agile methods are a subset of iterative and evolutionary methods [32, 33] and are based on iterative enhancement [5] and opportunistic development processes [18]. In all iterative products, each iteration is a self-contained, mini-project with activities that span requirements analysis, design, implementation, and test [32]. Each iteration leads to an iteration release (which may be only an internal release) that integrates all software across the team and is a growing and evolving subset of the final system. The purpose of having short iterations is so that feedback from iterations N and earlier, and any other new information, can lead to refinement and requirements adaptation for iteration N + 1. The customer adaptively specifies his or her requirements for the next release based on observation of the evolving product, rather than speculation at the start of the project [12]. There is quantitative evidence that frequent deadlines reduce the variance of a software process and, thus, may increase its predictability and efficiency.[40]

The pre-determined iteration length serves as a timebox for the team. Scope is chosen for each iteration to fill the iteration length. Rather than increase the iteration length to fit the chosen scope, the scope is reduced to fit the iteration length. A key difference between agile methods and past iterative methods is the length of each iteration. In the past, iterations might have been three or six months long. With agile methods, iteration lengths vary between one to four weeks, and intentionally do not exceed 30 days. Research has shown that shorter iterations have lower complexity and risk, better feedback, and higher productivity and success rates [32].

A point of commonality for all agile methods is the recognition of software development as an empirical process. In engineering, processes are classified as defined (also known as prescriptive) or empirical (or exploration-based) [36]. A defined process is one that can be started and allowed to run to completion, producing results with little variation each time [41]. Assembling an automobile is such a process. Engineers can design an unambiguous process to assemble a car and specify an assembly order and actions on the part of the assembly-line workers, machines, and robots. Generally speaking, if the manufacturing process follows these predefined steps, a high-quality car can be produced provided that the software process is under control and tolerances are appropriate, i.e., process variance is low. Such defined manufacturing processes are suitable for building items with a low degree of novelty or change and with a highly repeatable process [32].

Software development often has too much change during the time that the team is developing the product to be considered a defined process. A set of predefined steps may not lead to a desirable, predictable outcome because software development is a decidedly human activity: requirements change, technology changes, people are added and taken off the team, and so on. In other words, the process variance is high. In an engineering context, empirical processes are used for research-oriented, high-change, possibly unstable, and intellectually-intensive domains [32] that require constant monitoring and exploratory work [18]. These conditions necessitate short "inspect-and-adapt" cycles and frequent, short feedback loops [22, 36]. The short inspect-and-adapt cycles prominent in agile methodologies can help development teams to better handle the conflicting and unpredictable demands of some projects.

Another area of commonality among all agile methodologies is the importance of the people performing the roles and the recognition that, more so than any process or tool, these people are the most influential factoring in any project. Brooks' acknowledges the same in *The Mythical Man Month* [15], "The quality of the people on a project, and their organization and management, are more important factors in success than are the tools they use or the technical approaches they take." Unfortunately, there are commonalities among some agile methods that may be less than positive. One is that, unlike more classical iterative methods, explicit quantitative quality measurements and process modeling and metrics are often subdued and sometimes completely avoided. Possible justifications for this lack of modeling and metrics range from lack of time, to lack of skills, to intrusiveness, to social reasons.

Another potential problem area for agile methods is the ability to cope with corrections or deficiencies introduced into the product. Ideally, even in "classical" development environments, the reaction to change information need be quick; the corrections are applied within the same life-cycle phase in which the information is collected. However, introduction of feedback loops into the software process will depend on the software engineering capabilities of the organization, and the reaction latency will depend on the accuracy of the feedback models. For example, it is unlikely that organizations below the third maturity level on the Software Engineering Institute (SEI) Capability Maturity Model (CMM) scale [38] would have processes that could react to the feedback information in less than one software release cycle. This needs to be taken into account when considering the level and the economics of the "agile" methods. For instance, only relatively small teams can self-organize (one of the agile principles) into something resembling a CMM Level 4 or 5 performance. Also, since personnel resources are not unlimited, there is also some part of the software that may go untested, or may be verified to a lesser degree. The basic, and most difficult, aspect of system verification is to decide what must be tested, and what can be left untested, or partially tested [43].

1.2 Agile Development and Principles

In February 2001, several software engineering consultants joined forces and began to classify a number of similar change-sensitive methodologies as *agile* (a term with a decade of use in flexible manufacturing practices [34] which began to be used for software development in the late 1990's [3]). The term promoted the professed ability for rapid and flexible response to change of the methodologies. The consultants formed the Agile Alliance and wrote The Manifesto for Agile Software Development and the Principles Behind the Agile Manifesto [9, 25]. The methodologies originally embraced by the Agile Alliance were Adaptive Software Development (ASD) [26], Crystal [17], Dynamic Systems Development Method (DSDM) [42], Extreme Programming (XP) [6], Feature Driven Development (FDD) [16, 37] and Scrum [41].

1.2.1 Agile Software Development

The Agile Alliance documented its value statement [9] as follows:.

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

Individuals and interactions   over   processes and tools

Working software	*over*	*comprehensive documentation*
Customer collaboration	*over*	*contract negotiation*
Responding to change	*over*	*following a plan*

That is, while there is value in the items on the right, we value the items on the left more.

The implication is that formalization of the software process hinders the human and practical component of software development, and thus reduces the chance for success. While this is true when formalization is misused and misunderstood, one has to be very careful not to overemphasize and under-measure the items on the left hand side since this can lead to the same problem, poor quality software. The key is appropriate balance [13].

1.2.2 The Principles

The Agile Alliance also documented the principles they follow that underlie their manifesto [9]. As such the agile methods are principle-based, rather than rule-based [32]. Rather than have predefined rules regarding the roles, relationships, and activities, the team and manager are guided by these principles:

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 time scale.*
4. *Business people and developers must work together daily through 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.*
9. *The sponsors, developers, and users should be able to maintain a constant pace indefinitely.*
10. *Continuous attention to technical excellence and good design enhances agility.*
11. *Simplicity – the art of maximizing the amount of work not done – is essential.*
12. *The best architectures, requirements, and designs emerge from self-organizing teams.*
13. *At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.*

2 Examples of Agile Software Development Methodologies

This section provides a brief introduction to three agile methodologies. The three were chosen to demonstrate the range of applicability and specification of the agile methodologies. For each methodology we provide an overview, and then discuss documents and artifacts produced by the development team, the roles the members of the development team assume, and the process.

2.1 Extreme Programming (XP)

Extreme Programming (XP) [6] originators aimed at developing a methodology suitable for "object-oriented projects using teams of a dozen or fewer programmers in one location." [29]

The methodology is based upon five underlying values: communication, simplicity, feedback, courage, and respect.

- **Communication**. XP has a culture of oral communication and its practices are designed to encourage interaction. The communication value is based on the observation that most project difficulties occur because someone *should have* spoken with someone else to clarify a question, collaborate, or obtain help. "Problems with projects can invariably be traced back to somebody not talking to somebody else about something important." [6]
- **Simplicity**. Design the simplest product that meets the customer's needs. An important aspect of the value is to only design and code what is in the current requirements rather than to anticipate and plan for unstated requirements.
- **Feedback**. The development team obtains feedback from the customers at the end of each iteration and external release. This feedback drives the next iteration. Additionally, there are very short design and implementation feedback loops built into the methodology via pair programming and test-driven development [44].
- **Courage**. The other three values allow the team to have courage in its actions and decision making. For example, the development team might have the courage to resist pressure to make unrealistic commitments.
- **Respect**. Team members need to care about each other and about the project.

2.1.1 Documents and Artifacts

In general, XP relies on "documentation" via oral communication, the code itself, and tacit knowledge transfer rather than written documents and artifacts. However, while oral communication may work for small groups, it is not a recommended procedure for large systems, high-risk systems, or systems that require audit-ability for legal or software reliability engineering reasons. In these cases, the following "tools" may need to be more formally managed, recorded/preserved and regularly re-visited as part of a more "formal" and traceable XP process.

.

- **User story cards**, paper index cards which contain brief requirement (features, fixes, non-functional) descriptions. The user story cards are intentionally not a full requirement statement but are, instead, a commitment for further conversation between the developer and the customer. During this conversation, the two parties will come to an oral understanding of what is needed for the requirement to be fulfilled. Customer priority and developer resource estimate are added to the card. The resource estimate for a user story must not exceed the iteration duration.

- **Task list**, a listing of the tasks (one-half to three days in duration) for the user stories that are to be completed for an iteration. Tasks represent concrete aspects of a user story. Programmers volunteer for tasks rather than are assigned to tasks.
- **CRC cards [10] (optional)**, paper index card on which one records the responsibilities and collaborators of classes which can serve as a basis for software design. The classes, responsibilities, and collaborators are identified during a design brainstorming/role-playing session involving multiple developers. CRC stands for **C**lass-**R**esponsibility-**C**ollaboration.
- **Customer acceptance tests**, textual descriptions and automated test cases which are developed by the customer. The development team demonstrates the completion of a user story and the validation of customer requirements by passing these test cases.
- **Visible Wall Graphs**, to foster communication and accountability, progress graphs are usually posted in team work area. These progress graphs often involve how many stories are completed and/or how many acceptance test cases are passing.

2.1.2 Roles

- **Manager,** owns the team and its problems. He or she forms the team, obtain resources, manage people and problems, and interfaces with external groups.
- **Coach**, teaches team members about the XP process as necessary, intervenes in case of issues; monitors whether the XP process is being followed. The coach is typically a programmer and not a manager.
- **Tracker**, regularly collects user story and acceptance test case progress from the developers to create the visible wall graphs. The tracker is a programmer, not a manager or customer.
- **Programmer**, writes tests, design, and code; refactors; identifies and estimates tasks and stories (this person may also be a tester)
- **Tester**, helps customers write and develop tests (this person may also be a programmer)
- **Customer**, writes stories and acceptance tests; picks stories for a release and for an iteration. A common misconception is that the role of the customer must be played by one individual from the customer organization. Conversely, a group of customers can be involved or a customer representative can be chosen from within the development organization (but external to the development team).

2.1.3 Process

The initial version of the XP software methodology [6] published in 2000 had 12 programmer-centric, technical practices. These practices interact, counterbalance and reinforce each other [6, 27]. However, in a survey [20] of project managers, chief executive officers, developers, and vice-presidents of engineering for 21 software projects, it was found that none of the companies adopted XP in a “pure” form wherein all 12 practices were used without adaptation. In 2005, XP was changed to include 13 primary practices and 11 corollary practices [8]. The primary practices are intended to be [8] useful independent of each other and the other practices used, though the interactions between the practices may amplify their effect. The corollary practices are likely to be difficult without first mastering a core set of the primary practices.

Below the 13 primary technical practices of XP are briefly described:

- **Sit together,** the whole team develops in one open space.
- **Whole team,** utilize a cross-functional team of all those necessary for the product to succeed.
- **Informative workspace,** place visible wall graphs around the workspace so that team members (or other interested observers) can get a general idea of how the project is going.
- **Energized work**, XP teams do not work excessive overtime for long periods of time. The motivation behind this practice is to keep the code of high quality (tired programmers inject more defects) and the programmers happy (to reduce employee turnover). Tom DeMarco contends that, "Extended overtime is a productivity-reducing technique." [19]
- **Pair programming** [46], refers to the practice whereby two programmers work together at one computer, collaborating on the same design, algorithm, code, or test.
- **Stories**, the team write short statements of customer-visible functionality desired in the product. The developers estimate the story; the customer prioritizes the story.
- **Weekly cycle**, at the beginning of each week a meeting is held to review progress to date, have the customer pick a week's worth of stories to implement that week (based upon developer estimates and their own priority), and to break the stories into tasks to be completed that week. By the end of the week, acceptance test cases for the chosen stories should be running for demonstration to the customer to drive the next weekly cycle.
- **Quarterly cycle**, the whole team should pick a theme or themes of stories for a quarter's worth of stories. Themes help the team reflect on the bigger picture. At the end of the quarter, deliver this business value.
- **Slack,** in every iteration, plan some lower-priority tasks that can be dropped if the team gets behind such that the customer will still be delivered their most important functionality.
- **Ten-minute build,** structure the project and its associated tests such that the whole system can be built and all the tests can be run in ten minutes so that the system will be built and the tests will be run often.
- **Test-first programming,** all stories have at least one acceptance test, preferably automated. When the acceptance test(s) for a user story all pass, the story is considered to be fulfilled. Additionally, automated unit tests are incrementally written using the test-driven development (TDD) [7] practice in which code and automated unit tests are alternately and incrementally written on a minute-by-minute basis.
- **Continuous integration**, programmers check in to the code base completed code and its associated tests several times a day. Code may only be checked in if all its associated unit tests and all of unit tests of the entire code base pass.
- **Incremental design**, rather than develop an anticipatory detailed design prior to implementation, invest in the design of the system every day in light of the experience of the past. The viability and prudence of anticipatory design has

changed dramatically in our volatile business environment [27]. Refactoring [24] to improve the design of previously-written code is essential. Teams with robust unit tests can safely experiment with refactorings because a safety net is in place.

Below the 11 corollary technical practices of XP are briefly described:

- **Real customer involvement**, the customer is available to clarify requirements questions, is a subject matter expert, and is empowered to make decisions about the requirements and their priority. Additionally, the customer writes the acceptance tests.
- **Incremental deployment,** gradually deploy functionality in a live environment to reduce the risk of a big deployment.
- **Team continuity,** keep effective teams together.
- **Shrinking team,** as a team grows in capacity (due to experience), keep their workload constant but gradually reduce the size of the team.
- **Root cause analysis,** examine the cause of a discovered defect by writing acceptance test(s) and unit test(s) to reveal the defect. Subsequently, examine why the defects was created but not caught in the development process.
- **Shared code**, once code and its associated tests are checked into the code base, the code can be altered by any team member. This collective code ownership provides each team member with the feeling of owning the whole code base and prevents bottlenecks that might have been caused if the "owner" of a component was not available to make a necessary change.
- **Code and tests,** maintain only the code and tests as permanent artifacts. Rely on social mechanisms to keep alive the important history of the project.
- **Daily deployment,** put new code into production every night.
- **Negotiated scope contract,** fix the time, cost, and required quality of a project but call for an on-going negotiation of the scope of the project.
- **Pay-per-use,** charge the user every time the system is used to obtain their feedback by their usage patterns.

Though not one of the "official" XP practices, essentially all XP teams also have short **Stand-Up Meetings** daily [4]. In these meetings, the team stands in a circle (standing is intentional to motivate the team to keep the meeting short). In turn, each member of the team tells the group:

- What he or she accomplished the prior day
- What he or she plans to do today
- Any obstacles or difficulties he or she is experiencing

Often the pair-programming pairs are dynamically formed during the daily meeting as the tasks for the day are discussed and the two programmers that are best equipped to handle the task join together.

A "courageous" XP manager will keep a record of such meetings in order to turn them into quantitative progress measures of the project [21, 35, 43]. The burden of this quantification may be eased and automated through appropriate tools.

2.2 Crystal

The Rational Unified Process (RUP) [30, 31] is a customizable process framework. Depending upon the project characteristics, such as team size and project size, the RUP can be tailored or extended to match the needs of an adopting organization. Similarly, the family of Crystal Methods [17] were developed to address the variability of the environment and the specific characteristics of the project. However, RUP generally starts with a plan-driven base methodology and tailors down for smaller, less critical projects. Conversely, Crystal author Alistair Cockburn feels that the base methodology should be "barely sufficient." He contends, "You need one less notch control than you expect, and less is better when it comes to delivering quickly." [27] Moreover, since the project and the people evolve over time, the methodology so too must be tuned and evolved during the course of the project.

Crystal is a family of methods because Cockburn believes that there is no "one-size-fits-all" development process. As such, the different methods are assigned colors arranged in ascending opacity; the most agile version is Crystal Clear, followed by Crystal Yellow, Crystal Orange, and Crystal Red. The graph in Figure 2 is used to aid the choice of a Crystal Method starting point (for later tailoring). Along the x-axis is the size of the team. As a team gets larger (moves to the right along the x-axis), the harder it is to manage the process via face-to-face communication and, thus, the greater the need for coordinating documentation, practices, and tools. The y-axis addresses the system's potential for causing damage. The lowest damage impact is loss of comfort, then loss of discretionary money, loss of essential money, and finally loss of life. Based on the team size and the criticality, the corresponding Crystal methodology is identified. Each methodology has a set of recommended practices, a core set of roles, work products, techniques, and notations.

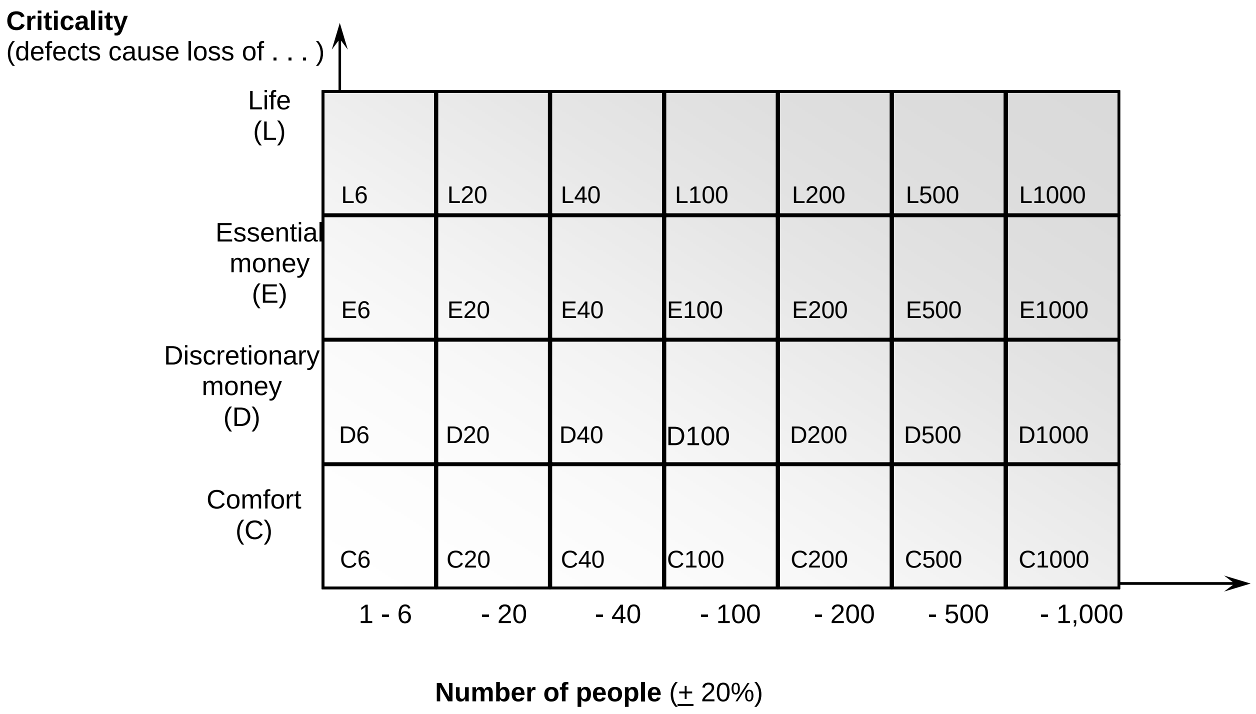

Figure 1: The Family of Crystal Methods [adapted from [17]]

All the Crystal Methods emphasize the importance of people in developing software. "[Crystal] focuses on people, interaction, community, skills, talents, and communication as first order effects on performance. Process remains important, but secondary". [27] There are only two absolute rules of the Crystal family of methodologies. First, incremental cycles must not exceed four months. Second, reflection workshops must be held after every delivery so that the methodology is self-adapting. Currently, only Crystal Clear and Crystal Orange have been defined. Summaries of these two methodologies are provided below.

2.2.1 Crystal Clear

Crystal Clear [17] is targeted at a D6 project and could be applied to a C6 or a E6 project and possibly to a D10 project. Crystal Clear is an optimization of Crystal that can be applied when the team consists of three to eight people sitting in the same room or adjoining offices. The property of close communication is strengthened to "osmotic" communication meaning that people overhear each other discussing project priorities, status, requirements, and design on a daily basis. Crystal Clear's model elements are as follows:

- **Documents and artifacts**: release plan, schedule of reviews, informal/low-ceremony use cases, design sketches, running code, common object model, test cases, and user manual
- **Roles**: project sponsor/customer, senior designer-programmer, designer-programmer, and user (part time at least)
- **Process:** incremental delivery, releases less than two to three months, some automated testing, direct user involvement, two user reviews per release, and methodology-tuning retrospectives. Progress is tracked by software delivered or major decisions reached, not by documents completed.

2.2.2 Crystal Orange

Crystal Orange is targeted at a D40 project. Crystal Orange is for 20-40 programmers, working together in one building on a project in which defects could cause the loss of discretionary money (i.e., medium risk). The project duration is between one and two years and time-to-market is important. Crystal Clear's model elements are as follows:

- **Documents and artifacts**: requirements document, release plan, schedule, status reports, UI design document, inter-team specs, running code, common object model, test cases, migration code, and user manual
- **Roles**: project sponsor, business expert, usage expert, technical facilitator, business analyst, project manager, architect, design mentor, lead designer-programmer, designer-programmer, UI designer, reuse point, writer, and tester
- **Process:** incremental delivery, releases less than three to four months, some automated testing, direct user involvement, two user reviews per release, and methodology-tuning retrospectives.

2.3 Scrum

2.3.1 Overview

In the Scrum process [27, 41] puts a project management “wrapper” around a software development methodology. The methodology is flexible on how much/how little ceremony but the Scrum philosophy would guide a team towards as little ceremony as possible. Usually a Scrum teams works co-located. However, there have been Scrum teams that work geographically distributed whereby team members participate in daily meeting via speakerphone. Scrum teams are self-directed and self-organizing teams. The team commits to a defined goal for an iteration and is given the authority, autonomy, and responsibility to decide how best to meet it.

2.3.2 Documents and Artifacts

There are three main artifacts produced by Scrum teams, the Product Backlog, the Sprint Backlog, and the Sprint Burndown chart. All of these are openly accessible and intentionally visible to the Scrum Team.

- **Product Backlog**, an evolving, prioritized queue of business and technical functionality that needs to be developed into a system and defects that need to be fixed [41] during the release. For each requirement, the Product Backlog contains a unique identifier for the requirement, the category (feature, enhancement, defect), the status, the priority, and the estimate for the feature. It is kept in a spreadsheet-like format.
- **Sprint Backlog**, a list of all business and technology features, enhancements, and defects that have been scheduled for the current iteration (called a Sprint). The Sprint Backlog is also maintained in a spreadsheet-like format. The requirements are broken down into tasks. For each task in the backlog, the spreadsheet contains a short task description, who originated the task, who owns the task, the status and the number of hours remaining to complete the task. The Sprint Backlog is updated each day by a daily tracker who visits the team members to obtain the latest estimates of the work remaining to complete the task. Estimates can increase when the team member realizes that the work was underestimated.
- **Sprint Burndown chart**. The hours remaining to complete Sprint tasks are graphed and predominantly displayed for the team. A sample of a burndown chart is shown in Figure 3.

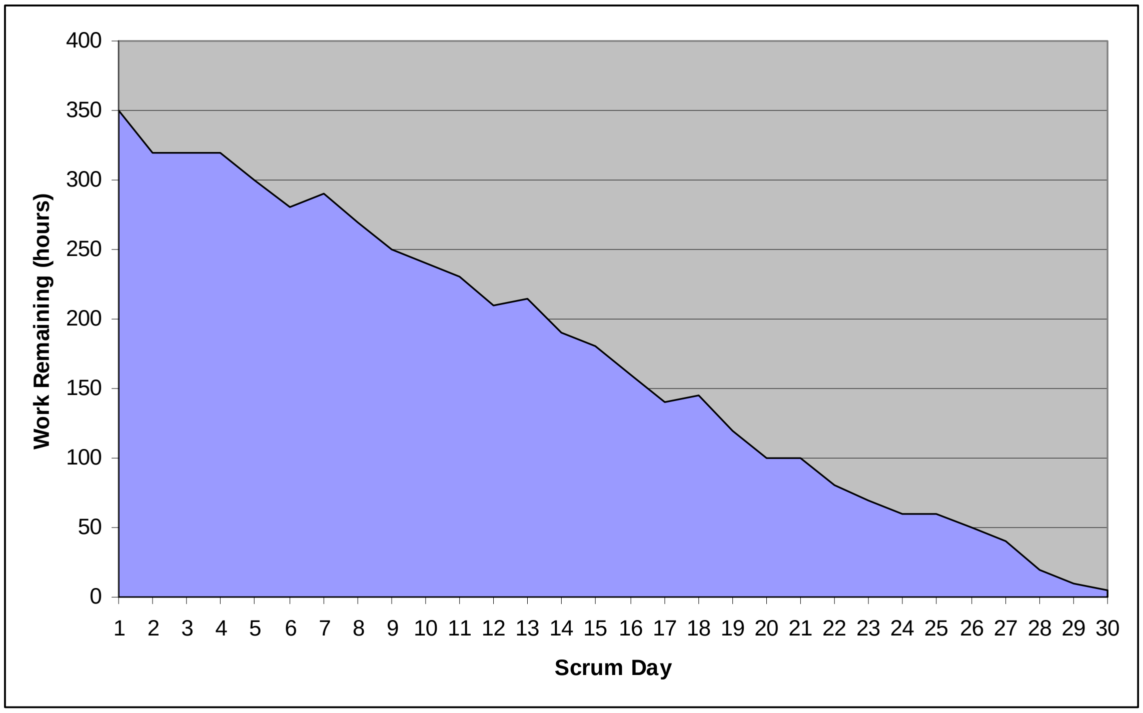

Figure 3: Sprint Burndown Chart

2.3.3 Roles

- **Product Owner**, the person who is responsible for creating and prioritizing the Product Backlog, choosing what will be included in the next iteration/Sprint, and reviewing the system (with other stakeholders) at the end of the Sprint.
- **Scrum Master,** knows and reinforces the product iteration and goals and the Scrum values and practices, conducts the daily meeting (the Scrum Meeting) and the iteration demonstration (the Sprint Review), listens to progress, removes impediments (blocks), and provides resources. The Scrum Master is also a Developer (see below) and participates in product development (is not just management).
- **Developer,** member of the Scrum team. The Scrum Team is committed to achieving a Sprint Goal and has full authority to do whatever it takes to achieve the goal. The size of a Scrum team is seven, plus or minus two.

2.3.4 Process

An overview of the Scrum process is provided in Figure 4. Each of the elements of the process will be discussed in this section.

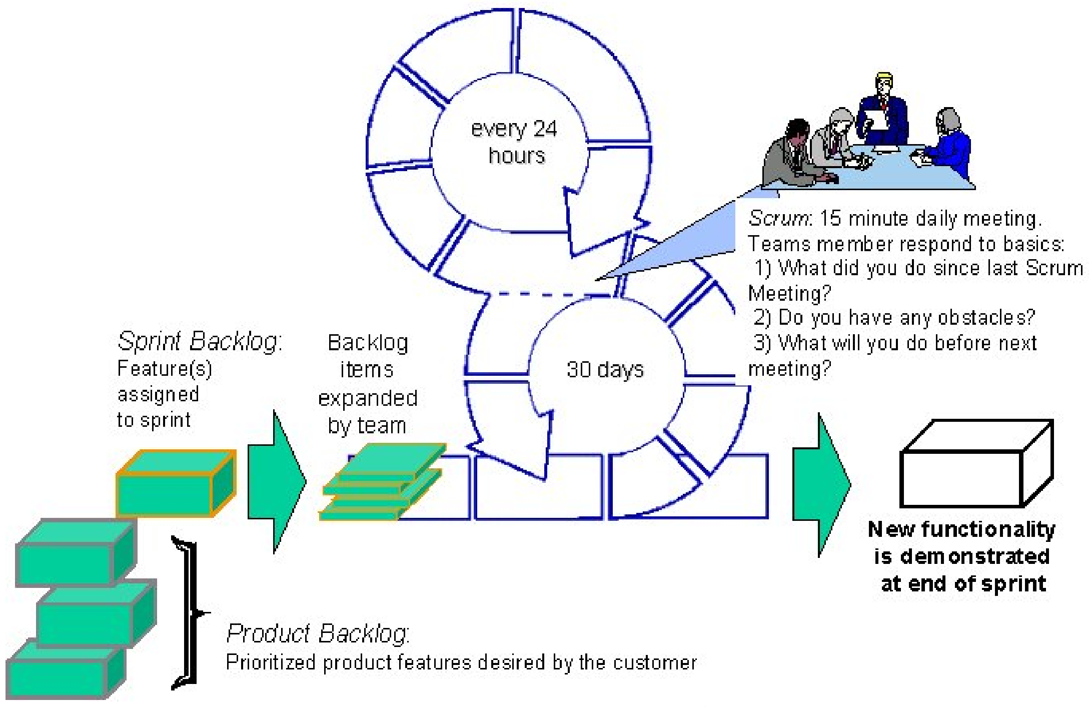
every 24 hours
Scrum: 15 minute daily meeting.
Teams member respond to basics:
1) What did you do since last Scrum Meeting?
2) Do you have any obstacles?
3) What will you do before next meeting?
Sprint Backlog: Feature(s) assigned to sprint
Backlog items expanded by team
30 days
New functionality is demonstrated at end of sprint
Product Backlog: Prioritized product features desired by the customer

Figure 4: The Scrum Process (from [1])

The Scrum process is composed of the following:

- A *Sprint Planning* meeting is held with the development team, management, and the Product Owner. The Product Owner is a representative of the customer or a contingent of customers. The Product Owner creates and prioritizes the Product Backlog. In the planning meeting, the Product Owner chooses which features are included in the next 30-day increment (called a Sprint) usually driven by highest business value and risk. Additionally a Sprint Goal is established which serves as a minimum, high-level success criteria for the Sprint and keep the Scrum Team focused on the big picture, rather than just on the chosen features. The development team figures out the tasks and resources required to deliver those features. Jointly, they determine a reasonable number of features to be included in the next Sprint. Once this set of features has been identified, no re-prioritization takes place during the ensuing 30-day Sprint in which features are designed, implemented and tested.
- During a Sprint, code is integrated and regression tested daily.
- Short, 15-minute *Scrum Meetings* are held daily. These meetings are similar to XP Stand Up Meetings described above because XP's meetings are based on the success Scrum had with its Scrum Meetings. The meeting is held in a room with a whiteboard so that tasks and blocks can be written down. While others (such as managers) may attend the Sprint Meeting, only the team members and the Scrum Master can speak. Each team member answers the following questions:
    - What have you done since the last Scrum?
    - What will you do between now and the next Scrum?
    - What got in your way of doing work?

The Scrum Meeting is an essential component of the methodology. Social promises are made in the meeting which seems to increase responsibility and follow-through and to keep the project on course. However, these meetings can become unmanageable if they are run with too many people; it is recommended that each team has a maximum of seven members. For use with larger teams, the team subdivides into smaller groups, each having its own Scrum meeting. One representative from each of the smaller groups attends a "Scrum of Scrums" meeting. This representative answers the Scrum questions, highlighting the activities of his or her own sub-team. In this way, essential information is passed between sub-teams.

- At the end of a Sprint, a *Sprint Review* takes place to review progress, demonstrate features to the customer, management, users and the Product Owner and review the project from a technical perspective. The meeting is conducted by the Scrum Master. The Product Owner and other interested stakeholders attend the meeting. The latest version of the product is demonstrated in which the functions, design, strength, weaknesses, and trouble spots are shared with the Product Owner. The focus is on showing the product itself; formal presentations (such as with PowerPoint slides) are forbidden.
- The cycle continues with a Sprint Planning meeting taking place to choose the features for the next Sprint.

2.4 Feature-Driven Development (FDD)

Feature Driven Development (FDD) [16, 37] authors Peter Coad and Jeff de Luca characterize the methodology as having "just enough process to ensure scalability and repeatability and encourage creativity and innovation all along the way." [27] Throughout, FDD emphasizes the importance of having good people and strong domain experts. FDD is build around eight best practices: domain object modeling; developing by feature; individual class ownership; feature teams; inspections; regular builds; configuration management; reporting/visibility of results. UML models [23] are used extensively in FDD.

2.4.1 Documents and Artifacts

- **Feature lists**, consisting of a set of features whereby features are small, useful in the eyes of the client, results; a client-valued function that can be implemented in two weeks or less. If a feature would take more than two weeks to implement, it must be further decomposed.
- **Design packages** consist of sequence diagrams and class diagrams and method design information
- **Track by Feature**, a chart which enumerates the features that are to be built and the dates when each milestone has been completed.
- **"Burn Up" Chart**, a chart that has dates (time) on the x axis. On the y axis is an increasing number of features that have been completed. As features are completed this chart indicates a positive slope over time.

2.4.2 Roles

- **Project manager**, is the administrative lead of the project responsible for reporting progress, managing budgets, and fighting for and managing resources including people, equipment, and space.
- **Chief architect**, is responsible for the overall design of the system including running workshop design sessions with the team.
- **Development manager**, is responsible for leading the day-to-day development activities including the resolution of resource conflicts.
- **Chief programmer**, as outlined by Brooks' ideas on surgical teams [15], is an experienced developer who acts as a team lead, mentor, and developer for a team of three to six developers. The chief programmer provides the breadth of knowledge about the skeletal model to a feature team, participates in high-level requirements analysis and design, and aids the team in low-level analysis, design, and development of new features.
- **Class owner**, is responsible for designing, coding, testing, and documenting new features in the classes that he or she owns.
- **Domain experts,** users, clients**,** sponsors, business analysts, etc. who have deep knowledge of the business for which the product is being developed.
- **Feature teams** are temporary groups of developers formed around the classes with which the features will be implemented. A feature team dynamically forms to implement a feature and disbands when the feature has been implemented (two weeks or less).

2.4.3 Process

The FDD process has five incremental, iterative processes, as shown in Figure 5. Guidelines are given for the amount of time that should be spent in each of these steps, constraining the amount of time spent in overall planning and architecture and emphasizing the amount of time designing and building features. Processes 1 through 3 are done at the start of a project and then updated throughout the development cycle. Processes 4 and 5 are done incrementally on two week cycles. Each of these processes has specific entry and exit criteria, whereby the entry criterion of Process N is the exit criteria of Process N-1.

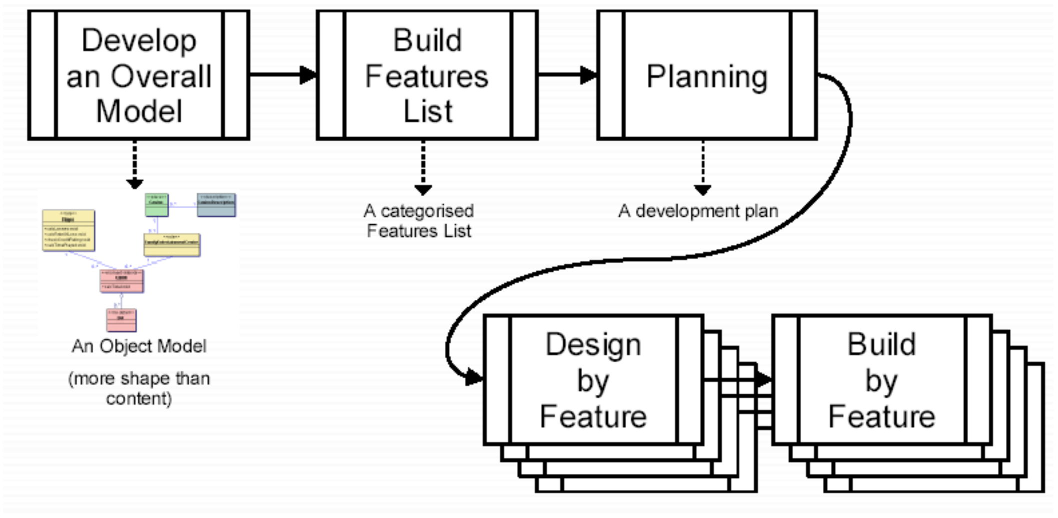

Figure 5: Overview of Feature Driven Development

- **Process 1: Develop an overall model** (time: 10 percent initially, 4 percent ongoing)
  Domain and development team members work together to understand the scope of the system and its context. High-level object models/class diagrams are developed for each area of the problem domain. Model notes record information about the model's shape and why some alternatives were selected and others rejected.
- **Process 2: Build a features list** (time: 4 percent initially, 1 percent ongoing)
  Complete list of all the features in the project; functional decomposition which breaks down a "business activity" requested by the customer to the features that need to be implemented in the software.
- **Process 3: Plan by feature** (time: 2 percent initially, 2 percent ongoing)
  A planning team consisting of the project manager, development manager, and chief programmer plan the order in which features will be developed. Planning is based on dependencies, risk, complexity, workload balancing, client-required milestones, and checkpoints. Business activities are assigned month/year completion dates. Every class is assigned to a specific developer. Features are bundled according to technical reasons rather than business reasons.
- **Process 4: Design by feature** (time 34% ongoing in two-week iterations)
  The chief programmer leads the development of design packages and refines object models with attributes. The sequence diagrams are often done as a group activity. The class diagrams and object models are done by the class owners. Domain experts interact with the team to refine the feature requirements. Designs are inspected.
- **Process 5: Build by feature** (time: 43% ongoing in two-week iterations)
  The feature team implements the classes and methods outlined by the design. This code is inspected and unit tested. The code is promoted to the build.

Progress is tracked and made visible during the Design by feature/Build by feature phases. Each feature has six milestones, three from the Design by feature phase (domain walkthrough, design, and design inspection) and three from the Build by feature phase (code, code inspection, promote to build). When these milestones are complete, the date is placed on the Track by Feature chart which is prominently displayed for the team. When a feature has completed all six milestones, this completion is reflected on the "Burn Up" chart. All features are scoped to be completed within a maximum of two weeks, including all six milestones.

3 Summary

In this chapter we presented an overview of the agile software development model and the characteristics of the projects that may be suited for the use of this model. Additionally, we provided overviews of three representative methodologies, XP, Crystal, and FDD. A summary of the distinguishing factors of these three methodologies is presented in Table 1.

Agile Methodology	Distinguishing Factor
Extreme Programming	• Intended for 10-12 co-located, object-oriented programmers • Four values • 12 highly-specified, disciplined development practices • Minimal archival documentation • Rapid customer and developer feedback loops
Crystal	• Customizable family of development methodologies for small to very large teams • Methodology dependent on size of team and criticality of project • Emphasis of face-to-face communication • Consider people, interaction, community, skills, talents, and communication as first-order effects • Start with minimal process and build up as absolutely necessary
Scrum	• Project management wrapper around methodology in which developer practices are defined • 30-day Sprints in which priorities are not changed • Daily Scrum meeting of Scrum Team • Burndown chart to display progress
Feature Driven Development	• Scalable to larger teams • Highly-specified development practices • Five sub-processes, each defined with entry and exit criteria • Development are architectural shape, object models and sequence diagrams (UML models used throughout) • Two-week features

Table 1: A summary of three agile software development methodologies

Several key ideas about agile methodologies presented in this chapter. The keys for successful use of choosing and using agile methodologies are summarized in Table 2.

🔑	Agile methods are a subset of iterative and evolutionary methods. Iterations are short to provide for more timely feedback to the project team.
🔑	The Agile Manifesto documents the priorities that underlie the principles and practices of agile software development methodologies.
🔑	Extreme Programming is based upon four values and 12 specific software development practices.
🔑	The Crystal family of methodologies is customizable based upon the characteristics of the project and the team.
🔑	Scrum mainly deals with project management principles. The methodology allows the team freedom to choose its specific development practices.
🔑	Of the four methodologies presented in the chapter, FDD has the most thorough analysis and design practices.

Table 2: Key Ideas for Agile Software Development Methodologies

There are other defined agile software development methodologies as well. These includes Adaptive Software Development (ASD) [26], Agile Modeling [2], Dynamic Systems Development Method (DSDM) [42], Lean Development [39], and Scrum [27, 41]. Additionally, teams can configure an agile RUP methodology. All agile methodologies consider software development to be an empirical process that necessitates short “inspect and adapt” feedback loops throughout the project.

References

[1] ADM Inc., *What is SCRUM*: http://controlchaos.com, 2004.
[2] S. W. Ambler, *Agile Modeling*. New York, NY: John Wiley and Sons, 2002.
[3] M. Aoyama, "Agile Software Process and its Experience," International Conference on Software Engineering, Kyoto, Japan, 1998, pp. 3-12.
[4] K. Auer and R. Miller, *XP Applied*. Reading, Massachusetts: Addison Wesley, 2001.
[5] V. R. Basili and A. J. Turner, "Iterative Enhancement: A Practical Technique for Software Development," *IEEE Transactions on Software Engineering*, vol. 1, no. 4, pp. 266 - 270, 1975.
[6] K. Beck, *Extreme Programming Explained: Embrace Change*. Reading, Mass.: Addison-Wesley, 2000.
[7] K. Beck, *Test Driven Development -- by Example*. Boston: Addison Wesley, 2003.
[8] K. Beck, *Extreme Programming Explained: Embrace Change*, Second ed. Reading, Mass.: Addison-Wesley, 2005.
[9] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas, "The Agile Manifesto," http://www.agileAlliance.org, 2001, 2001.
[10] D. Bellin and S. S. Simone, *The CRC Card Book*. Reading, Massachusetts: Addison-Wesley, 1997.
[11] B. Boehm, "Get Ready for Agile Methods, with Care," *IEEE Computer*, vol. 35, no. 1, pp. 64-69, 2002.
[12] B. Boehm, "A Spiral Model for Software Development and Enhancement," *Computer*, vol. 21, no. 5, pp. 61-72, May 1988.
[13] B. Boehm and R. Turner, "Using Risk to Balance Agile and Plan-Driven Methods," *IEEE Computer*, vol. 36, no. 6, pp. 57-66, June 2003.
[14] B. W. Boehm, *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1981.
[15] F. P. Brooks, *The Mythical Man-Month, Anniversary Edition*: Addison-Wesley Publishing Company, 1995.
[16] P. Coad, E. LeFebvre, and J. DeLuca, *Java Modeling in Color with UML*: Prentice Hall, 1999.
[17] A. Cockburn, *Agile Software Development*. Reading, Massachusetts: Addison Wesley Longman, 2001.
[18] B. Curtis, "Three Problems Overcome with Behavioral Models of the Software Development Process (Panel)," International Conference on Software Engineering, Pittsburgh, PA, 1989, pp. 398-399.
[19] T. DeMarco, *Slack: Getting Past Burnout, Busywork, and the Myth of Total Efficiency*: Broadway, 2002.
[20] K. El Emam, "Finding Success in Small Software Projects," *Agile Project Management*, vol. 4, no. 11, 2003.
[21] S. E. Elmaghraby, E. I. Baxter, and M. A. Vouk, "An Approach to the Modeling and Analysis of Software Production Processes," *Intl. Trans. Operational Res*, vol. 2, no. 1, pp. 117-135, 1995.
[22] R. Fairley, *Software Engineering Concepts*. New York: McGraw-Hill, 1985.
[23] M. Fowler, *UML Distilled*. Reading, Massachusetts: Addison Wesley, 2000.
[24] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Reading, Massachusetts: Addison Wesley, 1999.
[25] M. Fowler and J. Highsmith, "The Agile Manifesto," in *Software Development*, August 2001, pp. 28-32.

[26] J. Highsmith, *Adaptive Software Development*. New York, NY: Dorset House, 1999.
[27] J. Highsmith, *Agile Software Development Ecosystems*. Boston, MA: Addison-Wesley, 2002.
[28] W. S. Humphrey, *A Discipline for Software Engineering*. Reading, MA: Addison Wesley, 1995.
[29] R. Jeffries, A. Anderson, and C. Hendrickson, *Extreme Programming Installed*. Upper Saddle River, NJ: Addison Wesley, 2001.
[30] P. Kroll and P. Kruchten, *The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP*. Boston: Addison Wesley, 2003.
[31] P. Kruchten, *The Rational Unified Process: An Introduction*, Third ed. Boston: Addison Wesley, 2004.
[32] C. Larman, *Agile and Iterative Development: A Manager's Guide*. Boston: Addison Wesley, 2004.
[33] C. Larman and V. Basili, "A History of Iterative and Incremental Development," *IEEE Computer*, vol. 36, no. 6, pp. 47-56, June 2003.
[34] Lehigh University, "Agile Competition is Spreading to the World," http://www.ie.lehigh.edu/, 1991, 1991.
[35] N. Nagappan, L. Williams, and M. A. Vouk, "Towards a Metric Suite for Early Software Reliability Assessment," International Symposium on Software Reliability Engineering Fast Abstract, Denver, CO, 2003, pp.
[36] B. A. Ogunnaike and W. H. Ray, *Process Dynamics, Modeling, and Control*. New York: Oxford University Press, 1994.
[37] S. R. Palmer and J. M. Felsing, *A Practical Guide to Feature-Driven Development*. Upper Saddle River, NJ: Prentice Hall PTR, 2002.
[38] M. C. Paulk, B. Curtis, and M. B. Chrisis, "Capability Maturity Model for Software Version 1.1," Software Engineering Institute CMU/SEI-93-TR, February 24, 1993, 1993.
[39] M. Poppendieck and T. Poppendieck, *Lean Software Development*. Boston: Addison Wesley, 2003.
[40] T. Potok and M. Vouk, "The Effects of the Business Model on the Object-Oriented Software Development Productivity," *IBM Systems Journal*, vol. 36, no. 1, pp. 140-161, 1997.
[41] K. Schwaber and M. Beedle, *Agile Software Development with SCRUM*. Upper Saddle River, NJ: Prentice-Hall, 2002.
[42] J. Stapleton, *DSDM: The Method in Practice*, Second ed: Addison Wesley Longman, 2003.
[43] M. Vouk and A. T. Rivers, "Construction of Reliable Software in Resource-Constrained Environments," in *Case Studies in Reliability and Maintenance*, W. R. Blischke and D. N. P. Murthy, Eds. Hoboken, NJ: Wiley-Interscience, John Wiley and Sons, 2003, pp. 205-231.
[44] L. Williams, "The XP Programmer: The Few Minutes Programmer," *IEEE Software*, vol. 20, no. 3, pp. 16-20, May/June 2003.
[45] L. Williams and A. Cockburn, "Special Issue on Agile Methods," *IEEE Computer*, vol. 36, no. 3, June 2003.
[46] L. Williams and R. Kessler, *Pair Programming Illuminated*. Reading, Massachusetts: Addison Wesley, 2003.