NAME: LORO JOHN VIANNEY

REG NO.: J21B23/001

CHAPTER THREE

SOFTWARE PROCESS STRUCTURE

What exactly is a software process from a technical point of view?

A software process is a framework for the activities, actions, and tasks that are required to build high-quality software. A software process defines the approach that is taken as software is engineered. But software engineering also encompasses technologies that populate the process technical methods and automated tools.

Each of these activities, actions, and tasks resides within a framework or model that defines their relationship with the process and with one another. Each framework activity is populated by a set of software engineering actions. Each software engineering action is defined by a task set that identifies the work tasks that are to be completed, the work products that will be produced, the quality assurance points that will be required, and the milestones that will be used to indicate progress.

A GENERIC PROCESS MODEL

A generic process framework for software engineering defines five framework activities communication, planning, modeling, construction, and deployment. In addition, a set of umbrella activities project tracking and control, risk management, quality assurance, configuration management, technical reviews, and others are applied throughout the process.

A linear process flow executes each of the five framework activities in sequence, beginning with communication and culminating with deployment. **An iterative process flow** repeats one or more of the activities before proceeding to the next. **An evolutionary process flow** executes the activities in a “circular” manner. Each circuit through the five activities leads to a more complete version of the software. **A parallel process flow** executes one or more activities in parallel with other activities.

DEFINING A FRAMEWORK ACTIVITY

For a small software project requested by one person with simple, straightforward requirements, the communication activity might encompass little more than a phone call or email with the appropriate stakeholder. Therefore, the only necessary action is phone conversation, and the work tasks (the task set) that this action encompasses are:

- Make contact with stakeholder via telephone.
- Discuss requirements and develop notes.
- Organize notes into a brief written statement of requirements.
- Email to stakeholder for review and approval.

If the project was considerably more complex with many stakeholders, each with different set of (sometime conflicting) requirements, the communication activity might have six distinct actions: inception, elicitation, elaboration, negotiation, specification, and validation. Each of these software engineering actions would have many work tasks and a number of distinct work products.

IDENTIFYING A TASK SET

Each software engineering action can be represented by a number of different task sets each a collection of software engineering work tasks, related work products, quality assurance points, and project milestones.

PROCESS PATTERNS

A process pattern describes a process-related problem that is encountered during software engineering work, identifies the environment in which the problem has been encountered, and suggests one or more proven solutions to the problem. Patterns can be defined at any level of abstraction. In some cases, a pattern might be used to describe a problem (and solution) associated with a complete process model (e.g., prototyping). In other situations, patterns can be used to describe a problem (and solution) associated with a framework activity (e.g., planning) or an action within a framework activity (e.g., project estimating).

PROCESS ASSESSMENT AND IMPROVEMENT

The existence of a software process is no guarantee that software will be delivered on time, that it will meet the customer's needs, or that it will exhibit the technical characteristics that will lead to long-term quality characteristics. Process patterns must be coupled with solid software engineering practice. In addition, the process itself can be assessed to ensure that it meets a set of basic process criteria that have been shown to be essential for a successful software engineering. Approaches to software process assessment and improvement have been proposed over the past few decades:

Standard CMMI Assessment Method for Process Improvement (SCAMPI) provides a five-step process assessment model that incorporates five phases: initiating, diagnosing, establishing, acting, and learning. The SCAMPI method uses the SEI CMMI as the basis for assessment [SEI00].

CMM-Based Appraisal for Internal Process Improvement (CBA IPI) — provides a diagnostic technique for assessing the relative maturity of a software organization; uses the SEI CMM as the basis for the assessment [Dun01].

SPICE (ISO/IEC15504) —a standard that defines a set of requirements for software process assessment. The intent of the standard is to assist organizations in developing an objective evaluation of the efficacy of any defined software process [ISO08].

ISO 9001:2000 for Software— a generic standard that applies to any organization that wants to improve the overall quality of the products, systems, or services that it provides. Therefore, the standard is directly applicable to software organizations and companies [Ant06].

CHAPTER FOUR

PROCESS MODELS

Process models were originally proposed to bring order to the chaos of software development. History has indicated that these models have brought a certain amount of useful structure to software engineering work and have provided a reasonably effective road map for software teams. However, software engineering work and the products that are produced remain on “the edge of chaos.”

PRESCRIPTIVE PROCESS MODELS

A prescriptive process model 1 strives for structure and order in software development. Activities and tasks occur sequentially with defined guidelines for progress. Software process models can accommodate the generic framework activities described in Chapters 2 and 3, but each applies a different emphasis to these activities and defines a process flow that invokes each framework activity (as well as software engineering actions and tasks) in a different manner.

The Waterfall Model

The waterfall model, sometimes called the classic life cycle, suggests a systematic, sequential approach 2 to software development that begins with customer specification of requirements and progresses through planning, modeling, construction, and deployment, culminating in ongoing support of the completed software.

Incremental Process Models

The incremental model combines the elements’ linear and parallel process flows discussed in Chapter 3. The incremental model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces deliverable “increments” of the software.

Evolutionary Process Models

Evolutionary models are iterative. They are characterized in a manner that enables you to develop increasingly more complete versions of the software. In the paragraphs that follow, we present two common evolutionary process models.

Concurrent Models

The concurrent development model, sometimes called concurrent engineering, allows a software team to represent iterative and concurrent elements of any of the process models described in this chapter. Concurrent modeling defines a series of events that will trigger transitions from state to state for each of the software engineering activities, actions, or tasks. For example, during early stages of design (a major software engineering action that occurs during the modeling activity), an inconsistency in the requirements model is uncovered. This generates the event analysis model correction, which will trigger the requirements analysis action from the done state into the awaiting changes state.

SPECIALIZED PROCESS MODELS

Specialized process models take on many of the characteristics of one or more of the traditional models presented in the preceding sections. However, these models tend to be applied when a specialized or narrowly defined software engineering approach is chosen.

Component-Based Development

The component-based development model incorporates many of the characteristics of the spiral model. It is evolutionary in nature [Nie92], demanding an iterative approach to the creation of software. However, the component based development model comprises applications from prepackaged software components.

Modeling and construction activities begin with the identification of candidate components. These components can be designed as either conventional software modules or object-oriented classes or packages 11 of classes. Regardless of the technology that is used to create the components, the component-based development model incorporates the following steps (implemented using an evolutionary approach):

- Available component-based products are researched and evaluated for the application domain in question.
- Component integration issues are considered.
- Software architecture is designed to accommodate the components.
- Components are integrated into the architecture.
- Comprehensive testing is conducted to ensure proper functionality.

The Formal Methods Model

The formal methods model encompasses a set of activities that leads to formal mathematical specification of computer software. Formal methods enable you to specify, develop, and verify a computer-based system by applying a rigorous, mathematical notation. A variation on this approach, called cleanroom software engineering, is currently applied by some software development organizations.

Aspect-Oriented Software Development

Aspect-oriented software development (AOSD), often referred to as aspect-oriented programming (AOP) or aspect-oriented component engineering (AOCE) [Gru02], is a relatively new software engineering paradigm that provides a process and methodological approach for defining, specifying, designing, and constructing aspects —"mechanisms beyond subroutines and inheritance for localizing the expression of a crosscutting concern" [Elr01]. A distinct aspect-oriented process has not yet matured. However, it is likely that such a process will adopt characteristics of both evolutionary and concurrent process models.

THE UNIFIED PROCESS

The Unified Process is an attempt to draw on the best features and characteristics of traditional software process models, but characterize them in a way that implements many of the best principles of agile software development. The Unified Process recognizes the importance of customer communication and streamlined methods for describing the customer's view of a system (the use case). 13 It emphasizes the important role of software architecture and "helps the architect focus on the right goals, such as understandability, reliance to future changes, and reuse" [Jac99]. It suggests a process flow that is iterative and incremental, providing the evolutionary feel that is essential in modern software development.

PERSONAL AND TEAM PROCESS MODELS

The best software process is one that is close to the people who will be doing the work. If a software process model has been developed at a corporate or organizational level, it can be effective only if it is amenable to significant adaptation to meet the needs of the project team that is actually doing software engineering work. In an ideal setting, you would create a process that best fits your needs, and at the same time, meets the broader needs of the team and the organization.

Personal Software Process

The Personal Software Process (PSP) emphasizes personal measurement of both the work product that is produced and the resultant quality of the work product. In addition PSP makes the practitioner responsible for project planning (e.g., estimating and scheduling) and empowers the practitioner to control the quality of all software work products that are developed. The PSP model defines five framework activities: Planning, High-level design, High-level design review, Development, Postmortem.

Team Software Process

TSP defines the following framework activities: project launch, high-level design, implementation, integration and test, and postmortem. Like their counterparts in PSP (note that terminology is somewhat different), these activities enable the team to plan, design, and construct software in a disciplined manner while at the same time quantitatively measuring the process and the product. The postmortem sets the stage for process improvements. TSP makes use of a wide variety of scripts, forms, and standards that serve to guide team members in their work.

PROCESS TECHNOLOGY

Process technology tools allow a software organization to build an automated model of the process framework, task sets, and umbrella activities discussed in Chapter 3. The model, normally represented as a network, can then be analyzed to determine typical workflow and examine alternative process structures that might lead to reduced development time or cost.

PRODUCT AND PROCESS

If the process is weak, the end product will undoubtedly suffer. But an obsessive overreliance on process is also dangerous. People derive as much (or more) satisfaction from the creative process as they do from the end product. The duality of product and process is one important element in keeping creative people engaged as software engineering continues to evolve

CHAPTER FIVE

AGILE DEVELOPMENT

WHAT IS AGILITY ?

Agile software engineering combines a philosophy and a set of development guidelines. The philosophy encourages customer satisfaction and early incremental delivery of software; small, highly motivated project teams; informal methods; minimal software engineering work products; and overall development simplicity.

WHAT IS AN AGILE PROCESS ?

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

- 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.
- 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.
- Analysis, design, construction, and testing are not as predictable (from a planning point of view) as we might like.

Agility 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 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.

EXTREME PROGRAMMING

The XP Process

Extreme Programming uses an object-oriented approach as its preferred development paradigm and encompasses a set of rules and practices that occur within the context of four framework activities: planning, design, coding, and testing.

- **Planning.** The planning activity (also called the planning game) begins with listening —a requirements gathering activity that enables the technical members of the XP team to understand the business context for the software and to get a broad feel for required output and major features and functionality.
- **Design.** XP design rigorously follows the KIS (keep it simple) principle. A simple design is always preferred over a more complex representation. In addition, the design provides implementation guidance for a story as it is written—nothing less, nothing more. The design of extra functionality (because the developer assumes it will be required later) is discouraged. 3 XP encourages the use of CRC cards (Chapter 10) as an effective mechanism for thinking about the software in an object-oriented context.
- **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). 5 Once the unit test 6 has been created, the developer is better able to focus on what must be implemented to pass the test.
- **Testing**. 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 22) whenever code is modified (which is often, given the XP refactoring philosophy).

Industrial XP

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**. The IXP team ascertains whether all members of the project community (e.g., stakeholders, developers, management) are on board, have the proper environment established, and understand the skill levels involved. Project community. The IXP team determines whether the right people, with the right skills and training have been staged for the project. The “community” encompasses technologists and other stakeholders.
- **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.
- **Test-driven management**. An IXP team establishes a series of measurable “destinations” [Ker05] that assess progress to date and then defines mechanisms for determining whether or not these destinations have been reached.

- **Retrospectives**. An IXP team conducts a specialized technical review (Chapter 20) 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.
- **Continuous learning**. The IXP team is encouraged (and possibly, incented) to learn new methods and techniques that can lead to a higher-quality product.

OTHER AGILE PROCESS MODELS

Scrum

Scrum is an agile software development method that was conceived by Jeff Sutherland and his development team in the early 1990s. Scrum principles are consistent with the agile manifesto and are used to guide development activities within a process that incorporates the following framework activities: requirements, analysis, design, evolution, and delivery. Within each framework activity, work tasks occur within a process pattern (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.

Dynamic Systems Development Method

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. DSDM then defines three different iterative cycles:

- **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 the 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, the functional model iteration and the 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. DSDM can be combined with XP (Section 5.4) to provide a combination approach that defines a solid process

model (the DSDM life cycle) with the nuts and bolts practices (XP) that are required to build software increments.

Agile Modeling

Agile Modeling (AM) is a practice-based methodology for effective modeling and documentation of software-based systems. Simply put, Agile Modeling (AM) is a collection of values, principles, and practices for modeling software that can be applied on a software development project in an effective and light-weight manner. Agile models are more effective than traditional models because they are just barely good, they don't have to be perfect. Agile modeling adopts all of the values that are consistent with the agile manifesto. The agile modeling philosophy recognizes that an agile team must have the courage to make decisions that may cause it to reject a design and refactor. The team must also have the humility to recognize that technologists do not have all the answers and those business experts and other stakeholders should be respected and embraced.

Agile Unified Process

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 that enables a team to visualize the overall process flow for a software project. However, within each of the activities, the team iterates to achieve agility and to deliver meaningful software increments to end users as rapidly as possible. Each AUP iteration addresses the following activities:

- **Modeling**. UML representations of the business and problem domains are created. 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 3, 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 29) addresses change management, risk management, and the control of any persistent work products 12 that are produced by the team. Project management tracks and controls the progress of the team and coordinates team activities.

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

CHAPTER FIVE

HUMAN ASPECTS OF SOFTWARE ENGINEERING

Software engineering has an abundance of techniques, tools, and methods designed to improve both the software development process and the final product. However, software isn't simply a product of the appropriate technical solutions applied inappropriate technical ways. Software is developed by people, used by people, and supports interaction among people. As such, human characteristics, behavior, and cooperation are central to practical software development.

CHARACTERISTICS OF A SOFTWARE ENGINEER

- An effective software engineer has a sense of individual responsibility. This implies a drive to deliver on her promises to peers, stakeholders, and her management. It implies that she will do what needs to be done, when it needs to be done in an overriding effort to achieve a successful outcome.
- An effective software engineer has an acute awareness of the needs of other members of his team, of the stakeholders that have requested a software solution to an existing problem, and the managers who have overall control over the project that will achieve that solution. He is able to observe the environment in which people work and adapt his behavior to both the environment and the people themselves.
- An effective software engineer is brutally honest. If she sees a flawed design, she points out the flaws in a constructive but honest manner. If asked to distort facts about schedule, features, performance, or other product or project characteristics she opts to be realistic and truthful.
- An effective software engineer exhibits resilience under pressure. As we noted previously in this book, software engineering is always on the edge of chaos. Pressure (and the chaos that can result) comes in many forms—changes in requirements and priorities, demanding stakeholders or peers, an unrealistic or overbearing manager. But an effective software engineer is able to manage the pressure so that his performance does not suffer.
- An effective software engineer has a heightened sense of fairness. She gladly shares credit with her colleagues. She tries to avoid conflicts of interest and never acts to sabotage the work of others.
- An effective software engineer exhibits attention to detail. This does not imply an obsession with perfection, but it does suggest that he carefully considers the technical decisions he makes on a daily basis against broader criteria (e.g., performance, cost, quality) that have been established for the product and the project.
- Finally, an effective software engineer is pragmatic. She recognizes that software engineering is not a religion in which dogmatic rules must be followed, but rather a discipline that can be adapted based on the circumstances at hand.

THE PSYCHOLOGY OF SOFTWARE ENGINEERING

The following roles may be assigned explicitly or can evolve naturally.

- Ambassador —represents the team to outside constituencies with the intent of negotiating time and resources and gaining feedback from stakeholders.
- Scout —crosses the team's boundary to collect organizational information. "Scouting can include scanning about external markets, searching for new technologies, identifying relevant activities outside of the team and uncovering pockets of potential competition."
- Guard —protects access to the team's work products and other information artifacts.
- Sentry —controls the flow of information that stakeholders and others send to the team.
- Coordinator —focuses on communicating horizontally across the team and within the organization (e.g., discussing a specific design problem with a group of specialists within the organization).

THE SOFTWARE TEAM

An effective team should foster a sense of trust. Software engineers on the team should trust the skills and competence of their peers and their managers. The team should encourage a sense of improvement, by periodically reflecting on its approach to software engineering and looking for ways to improve their work. The most effective software teams are diverse in the sense that they combine a variety of different skill sets. Highly skilled technologists are complemented by members who may have less technical background but are more empathetic to the needs of stakeholders. But not all teams are effective and not all teams jell. In fact, many teams suffer from what Jackman calls "team toxicity." She defines five factors that "foster a potentially toxic team environment": (1) a frenzied work atmosphere, (2) high frustration that causes friction among team members, (3) a "fragmented or poorly coordinated" software process, (4) an unclear definition of roles on the software team, and (5) "continuous and repeated exposure to failure."

TEAM STRUCTURES

The "best" team structure depends on the management style of your organization, the number of people who will populate the team and their skill levels, and the overall problem difficulty. Constantine suggests four "organizational paradigms" for software engineering teams:

1. A closed paradigm structures a team along a traditional hierarchy of authority. Such teams can work well when producing software that is quite similar to past efforts, but they will be less likely to be innovative when working within the closed paradigm.
2. A random paradigm structures a team loosely and depends on individual initiative of the team members. When innovation or technological breakthrough is required, teams following the random paradigm will excel. But such teams may struggle when "orderly performance" is required.
3. An open paradigm attempts to structure a team in a manner that achieves some of the controls associated with the closed paradigm but also much of the innovation that occurs when using the random paradigm. Work is performed collaboratively, with heavy communication and consensus-based decision making the trademarks of

open paradigm teams. Open paradigm team structures are well suited to the solution of complex problems but may not perform as efficiently as other teams.

4. A synchronous paradigm relies on the natural compartmentalization of a problem and organizes team members to work on pieces of the problem with little active communication among themselves.

AGILE TEAMS

The Generic Agile Team

The small, highly motivated project team, also called an agile team, adopts many of the characteristics of successful software project teams discussed in the preceding section and avoids many of the toxins that create problems. However, the agile philosophy stresses individual (team member) competency coupled with group collaboration as critical success factors for the team.

The XP Team

Beck defines a set of five values that establish a foundation for all work performed as part of extreme programming (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 the agile team 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 metaphors

THE IMPACT OF SOCIAL MEDIA

Email, texting, and videoconferencing have become ubiquitous activities in software engineering work. But these communication mechanisms are really nothing more than modern substitutes or supplements for the face-to-face contact. Social media is different.

First, a social network is defined for a software project. Using the network, the software team can draw from the collective experience of team members, stakeholders, technologists, specialists, and other businesspeople who have been invited to participate in the network (if the network is private) or to any interested party (if the network is public). And it can do this whenever an issue, a question, or a problem arises. There are a number of different forms of social media and each has a place in software engineering work. It is very important to note that privacy and security issues should not be overlooked when using social media for software engineering work. Much of the work performed by software engineers may be proprietary to their employer and disclosure could be very harmful. For that reason, the distinct benefits of social media must be weighed against the treat of uncontrolled disclosure of private information.

SOFTWARE ENGINEERING USING THE CLOUD

Cloud computing provides a mechanism for access to all software engineering work products, artifacts, and project-related information. It runs everywhere and removes the device dependency that was once a constraint for many software projects. It allows members of a software team to conduct platform- independent, low-risk trials of new software tools and to provide feedback on those tools. It provides new avenues for distribution and testing of beta software. It provides the potential for improved approaches to content and configuration management.

Because cloud computing can accomplish these things, it has the potential to influence the manner in which software engineers organize their teams, the way they do their work, the manner in which they communicate and connect, and the way software projects are managed. Software engineering information developed by one team member can be instantly available to all team members, regardless of the platform others are using or their location.

COLLABORATION TOOLS

Tools are essential to collaboration among team members, enabling the facilitation, automation, and control of the entire development process. Adequate tool support is especially needed in global software engineering because distance aggravates coordination and control problems, directly or indirectly, through its negative effects on communication. Many of the tools used in a CDE are no different from the tools that are used to assist in the software engineering activities discussed in Parts 2, 3, and 4 of this book. But a worthwhile CDE also provides a set of services that are specifically designed to enhance collaborative work. These services include:

- A namespace that allows a project team to store all work products and other information in a manner that enhances security and privacy, allowing access only to authorized individuals.
- A calendar for coordinating meeting and other project events.
- Templates that enable team members to create work products that have a consistent look and structure.
- Metrics support that tracks each team member’s contributions in a quantitative manner.

GLOBAL TEAMS

In the software domain, globalization implies more than the transfer of goods and services across international boundaries. For the past few decades, an increasing number of major software products have been built by software teams that are often located in different countries. These global software development (GSD) teams have many of the characteristics of a conventional software team, but a GSD team has other unique challenges that include coordination, collaboration, communication, and specialized decision making. Approaches to coordination, collaboration, and communication have been discussed earlier in this chapter. Decision making on all software teams is complicated by four factors:

- Complexity of the problem.
- Uncertainty and risk associated with the decision.
- The law of unintended consequences (i.e., work-associated decision has an unintended effect on another project objective).
- Different views of the problem that lead to different conclusions about the way forward.