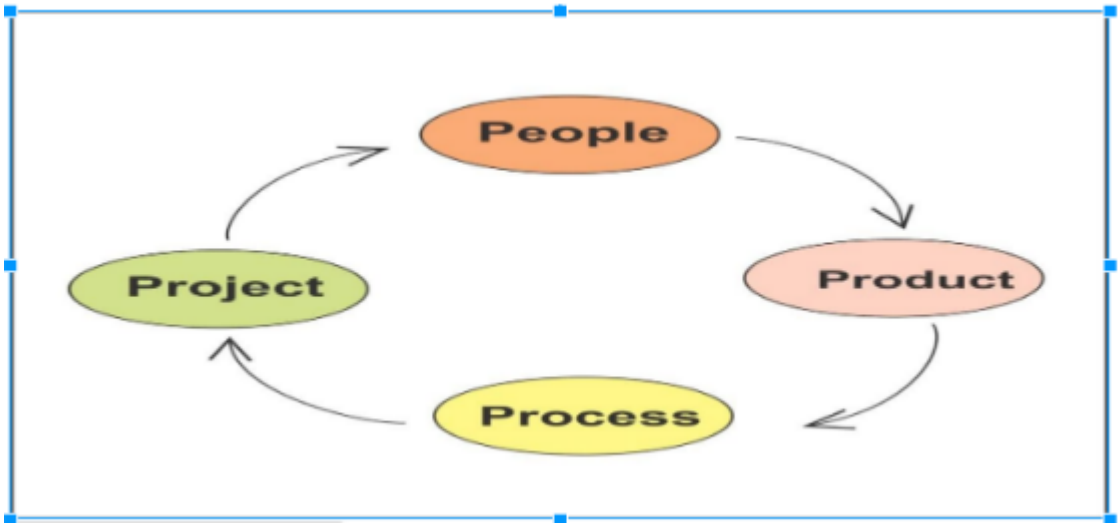
Unit 5

31.PROJECT MANAGEMENT CONCEPTS

1.“Effective software project management focuses on four P’s”. Justify this statement with suitable analogy.

2.Briefly explain the management spectrum in software project management.

31.1 THE MANAGEMENT SPECTRUM



Effective software project management focuses on the four Ps: people, product, process, and project. The order is not arbitrary. The manager who forgets that software engineering work is an intensely human endeavour will never have success in project management. A manager who fails to encourage comprehensive stakeholder communication early in the evolution of a product risks building an elegant solution for the wrong problem. The manager who pays little attention to the process runs the risk of inserting competent technical methods and tools into a vacuum. The manager who embarks without a solid project plan jeopardizes the success of the project.

31.1.1 The People

The cultivation of motivated, highly skilled software people has been discussed since the 1960s. In fact, the “people factor” is so important that the Software Engineering Institute has developed a People Capability Maturity Model (People-CMM), in recognition of the fact that “every organization needs to continually improve its ability to attract, develop, motivate, organize, and retain the workforce needed to accomplish its strategic business objectives” [Cur01].

The people capability maturity model defines the following key practice areas for software people: staffing, communication and coordination, work environment, performance management, training, compensation, competency analysis and development, career development, workgroup development, and team/ culture development, and others. Organizations that achieve high levels of People-CMM maturity have a higher likelihood of implementing effective software project management practices. The People-CMM is a

companion to the Software Capability Maturity Model– Integration (Chapter 37) that guides organizations in the creation of a mature software process. Issues associated with people management and structure for software projects are considered later in this chapter.

31.1.2 The Product

Before a project can be planned, product objectives and scope should be established, alternative solutions should be considered, and technical and management constraints should be identified. Without this information, it is impossible to define reasonable estimates of the cost, an effective assessment of risk, a realistic breakdown of project tasks, or a manageable project schedule that provides a meaningful indication of progress.

As a software developer, you and other stakeholders must meet to define product objectives and scope. In many cases, this activity begins as part of the system engineering or business process engineering and continues as the first step in software requirements engineering. Objectives identify the overall goals for the product (from the stakeholders' points of view) without considering how these goals will be achieved. Scope identifies the primary data, functions, and behaviours that characterize the product, and more important, attempts to bound these characteristics in a quantitative manner. Once the product objectives and scope are understood, alternative solutions are considered. Although very little detail is discussed, the alternatives enable managers and practitioners to select a "best" approach, given the constraints imposed by delivery deadlines, budgetary restrictions, personnel availability, technical interfaces, and myriad other factors.

31.1.3 The Process

A software process (Chapters 3–5) provides the framework from which a comprehensive plan for software development can be established. A small number of framework activities are applicable to all software projects, regardless of their size or complexity. A number of different task sets—tasks, milestones, work products, and quality assurance points—enable the framework activities to be adapted to the characteristics of the software project and the requirements of the project team. Finally, umbrella activities—such as software quality assurance, software configuration management, and measurement—overlay the process model. Umbrella activities are independent of any one framework activity and occur throughout the process.

31.1.4 The Project

We conduct planned and controlled software projects for one primary reason—it is the only known way to manage complexity. And yet, software teams still struggle. In a study of 250 large software projects between 1998 and 2004, Capers Jones [Jon04] found that "about 25 were deemed successful in that they achieved their schedule, cost, and quality objectives. About 50 had delays or overruns below 35 percent, while about 175 experienced major delays and overruns, or were terminated without completion." Although the success rate for present-day software projects may have improved somewhat, our project failure rate remains much

higher than it should be. 1 To avoid project failure, a software project manager and the software engineers who build the product must avoid a set of common warning signs, understand the critical success factors that lead to good project management, and develop a commonsense approach for planning, monitoring, and controlling the project. Each of these issues is discussed in Section 31.5 and in the chapters that follow.

Illustrate the roles of software teams and team leaders in software project management.

+

31.2 PEOPLE

The effectiveness of a software engineering team relies heavily on the people involved and their interactions within the software process. This process includes various stakeholders, each playing a crucial role in the success of the project. Here's a closer look at these stakeholders and how they contribute:

31.2.1 The Stakeholders

1. Senior Managers

- **Role:** Senior managers define the overarching business issues and strategic goals that often shape the direction and priorities of software projects.
- **Influence:** Their decisions on funding, resource allocation, and strategic focus can significantly impact the project's scope, schedule, and overall success.
- **Impact Example:** If senior managers prioritize a quick market release, the team may need to focus on delivering a minimum viable product (MVP) quickly, which might impact the depth of initial features but align with business goals.

2. Project (Technical) Managers

- **Role:** Project managers plan, motivate, organize, and control the practitioners doing the software work.
- **Responsibilities:** They are responsible for scheduling, resource management, risk management, and ensuring that the project meets its objectives.
- **Impact Example:** Effective project management can lead to better team coordination, timely project completion, and efficient handling of obstacles. Poor project management, on the other hand, can result in missed deadlines and project failure.

3. Practitioners

- **Role:** Practitioners are the technical professionals who carry out the actual software engineering tasks, including coding, testing, and maintenance.
- **Skills:** They bring the necessary technical expertise and problem-solving skills to develop high-quality software products.
- **Impact Example:** Well-trained and motivated practitioners can innovate and optimize processes, leading to better performance and quality. Conversely, lack of skills or motivation among practitioners can lead to subpar software quality and delays.

4. Customers

- **Role:** Customers specify the requirements for the software, providing the essential input needed to guide the development process.
- **Expectations:** They define what the software should do and what features it must include, directly influencing the design and functionality of the product.
- **Impact Example:** Clear and well-documented requirements from customers can lead to a product that meets expectations and adds value. Ambiguous or changing requirements can lead to scope creep and project delays.

5. End Users

- **Role:** End users are the individuals who interact with the software once it is deployed.
- **Feedback:** Their feedback is critical for continuous improvement and ensuring the software remains useful and relevant.
- **Impact Example:** Positive end-user feedback can lead to higher adoption rates and customer satisfaction, while negative feedback can highlight areas for improvement and potential issues that need addressing.

Integration of Stakeholders

Effective software engineering requires a collaborative effort among all these stakeholders. Here's how they work together:

- **Communication:** Regular and clear communication among stakeholders ensures that everyone is aligned with the project goals and understands their roles and responsibilities.
- **Feedback Loops:** Continuous feedback from customers and end users helps practitioners make necessary adjustments, leading to a better final product.

- **Decision-Making:** Senior managers provide strategic direction, while project managers handle day-to-day decisions that keep the project on track.
- **Skill Utilization:** Practitioners apply their technical skills to build the software, guided by the requirements specified by customers and influenced by end-user feedback.

31.2.2 Team Leaders

Project management, especially in software development, demands a unique blend of technical expertise and people skills. Competent practitioners may not always make effective team leaders due to a lack of necessary leadership qualities. This section explores the characteristics of effective team leaders and project managers through the MOI model of leadership and other key traits.

MOI Model of Leadership (Jerry Weinberg)

1. Motivation

- **Definition:** The ability to encourage technical people to perform at their best.
- **Application:** Effective leaders inspire their teams through a combination of pushing (direct encouragement) and pulling (creating an environment that naturally motivates).

2. Organization

- **Definition:** The ability to shape processes to translate initial concepts into final products.
- **Application:** Leaders must be adept at molding existing workflows or creating new ones that facilitate efficient and effective software development.

3. Ideas or Innovation

- **Definition:** The ability to foster creativity within the constraints of a given project.
- **Application:** Encouraging team members to think creatively and innovate, even when working within established boundaries, is crucial for successful project outcomes.

Key Traits of an Effective Project Manager (Edgemon)

1. Problem Solving:

- **Definition:** The capacity to diagnose technical and organizational issues, structure solutions, and adapt lessons from past projects.
- **Application:** A project manager must be flexible and able to pivot when initial solutions are ineffective, maintaining a problem-solving mindset throughout the project lifecycle.

2. Managerial Identity:

- **Definition:** The confidence to assume control when necessary and allow technical experts to exercise their instincts.
- **Application:** Balancing control with delegation is critical. A project manager must know when to step in and when to step back, enabling team members to utilize their expertise effectively.

3. Achievement:

- **Definition:** Rewarding initiative and accomplishment to optimize productivity.
- **Application:** Recognizing and rewarding team members' efforts and achievements fosters a culture of productivity and controlled risk-taking, which is essential for innovation and progress.

4. Influence and Team Building:

- **Definition:** The ability to read people, understand their needs, and maintain composure under stress.
- **Application:** An effective project manager must be adept at interpreting verbal and nonverbal cues, responding to team members' needs, and remaining calm in high-stress situations to keep the team focused and motivated.

Problem-Solving Management Style

Weinberg suggests that successful project leaders should adopt a problem-solving management style. This involves:

- **Understanding the Problem:** Focusing on diagnosing and comprehending the core issues to be solved.
- **Managing the Flow of Ideas:** Encouraging open communication and idea sharing within the team.
- **Emphasizing Quality:** Communicating the importance of quality through words and actions, ensuring that it is never compromised.

31.2.3 The Software Team

Effective organization of software teams is crucial for the success of software projects. This organization is influenced by several factors, and there are various paradigms to structure teams. Here’s a detailed look at these considerations and paradigms:

Factors to Consider in Team Structure

According to Mantei, the structure of a software engineering team should take into account the following factors:

1. **Difficulty of the Problem:** The complexity of the software problem to be solved.
2. **Size of the Program:** Measured in lines of code or function points.
3. **Team Lifetime:** The duration for which the team will remain together.
4. **Modularity of the Problem:** The extent to which the problem can be divided into smaller, manageable parts.
5. **Quality and Reliability Requirements:** The level of quality and reliability expected from the software.
6. **Delivery Date Rigidity:** The strictness of the project deadlines.
7. **Sociability Needs:** The level of communication and interaction required among team members.

Organizational Paradigms for Software Teams

Constantine suggests four organizational paradigms for structuring software engineering teams:

1. **Closed Paradigm:**
    - **Structure:** Traditional hierarchy of authority.
    - **Suitability:** Effective for projects similar to past efforts, but may stifle innovation.
    - **Example:** A team producing routine software updates might benefit from this structure.
2. **Random Paradigm:**
    - **Structure:** Loose organization relying on individual initiative.
    - **Suitability:** Excels in projects requiring innovation and technological breakthroughs, but may struggle with structured, orderly performance.
    - **Example:** A research and development team exploring new technologies might thrive under this paradigm.
3. **Open Paradigm:**
    - **Structure:** Balances control and innovation with collaborative work, heavy communication, and consensus-based decision-making.
    - **Suitability:** Well-suited for solving complex problems, though potentially less efficient.
    - **Example:** A team working on a multifaceted, innovative software solution might find this structure beneficial.
4. **Synchronous Paradigm:**
    - **Structure:** Organizes team members to work on different parts of the problem with minimal communication.

- **Suitability:** Works well when the problem is naturally compartmentalized.
- **Example:** A team developing different modules of a large software system independently.

Creating a High-Performance Team

For a software team to be effective, it must exhibit cohesiveness, often described as a "jelled" team. According to DeMarco and Lister, a jelled team is a group of people so strongly knit that the whole is greater than the sum of its parts. Characteristics of jelled teams include:

- **Common Goals:** Sharing a unified definition of success.
- **Team Spirit:** Exhibiting a strong team spirit and sense of eliteness.
- **Self-Motivation:** Functioning with minimal management and high motivation.
- **Increased Productivity:** Significantly more productive and motivated than average teams.

Team Toxicity

Jackman identifies five factors that contribute to a toxic team environment:

1. **Frenzied Work Atmosphere:** High-pressure environments that lead to burnout.
2. **High Frustration:** Frustration causing friction among team members.
3. **Fragmented Processes:** Poorly coordinated software processes.
4. **Unclear Roles:** Lack of clear role definitions on the team.
5. **Continuous Failure Exposure:** Frequent exposure to failure, leading to demotivation.

Avoiding Team Toxicity

Project managers can help avoid these toxic factors by:

- **Providing Necessary Information:** Ensuring the team has all the required information.
- **Defining Goals Clearly:** Keeping major goals and objectives stable unless changes are absolutely necessary.
- **Empowering Decision Making:** Giving the team responsibility for decision making.
- **Choosing Appropriate Processes:** Understanding the product and people, and allowing the team to select the process model.
- **Establishing Accountability:** Implementing mechanisms for accountability, such as technical reviews.
- **Using Team-Based Feedback:** Establishing techniques for feedback and problem solving.

Recognizing Human Differences

Teams often struggle due to differing human traits. Understanding and managing these differences is essential for creating cohesive teams:

- **Extroverts vs. Introverts:** Different approaches to communication and interaction.
- **Intuitive vs. Logical Information Processing:** Different methods of gathering and organizing information.
- **Decision-Making Styles:** Varying preferences for making decisions based on logic or intuition.
- **Work Style Preferences:** Different attitudes toward schedules and task closure.
- **Stress Responses:** Varied responses to deadlines and stress.

32.PROCESS AND PROJECT METRICS

32.1 METRICS IN THE PROCESS AND PROJECT DOMAINS

Process metrics are collected across all projects and over long periods of time. Their intent is to provide a set of process indicators that lead to long-term software process improvement. On the other hand, **project metrics** enable a software project manager to:

1. Assess the status of an ongoing project.
2. Track potential risks.
3. Uncover problem areas before they go "critical."
4. Adjust workflow or tasks.
5. Evaluate the project team's ability to control the quality of software work products.

Measures collected by a project team and converted into metrics for use during a project can also be transmitted to those responsible for software process improvement. Hence, many of the same metrics are used in both the process and project domains.

32.1.1 Process Metrics and Software Process Improvement

The only rational way to improve any process is to measure specific attributes of the process, develop a set of meaningful metrics based on these attributes, and then use the metrics to provide indicators that will lead to a strategy for improvement.

Process is just one of several controllable factors in improving software quality and organizational performance. Referring to Figure 32.1, process sits at the center of a triangle connecting three factors that have a profound influence on software quality and organizational performance:

1. The skill and motivation of people.
2. The complexity of the product.
3. The technology (software engineering methods and tools) used.

Additionally, the process triangle exists within a circle of environmental conditions, including the development environment, business conditions, and customer characteristics.

You can only measure the efficacy of a software process indirectly by deriving a set of metrics based on process outcomes, which include:

- Errors uncovered before software release.
- Defects delivered to and reported by end users.
- Work products delivered (productivity).
- Human effort expended.
- Calendar time used.
- Schedule conformance.

You can also derive process metrics by measuring the characteristics of specific software engineering tasks, such as the effort and time spent performing umbrella activities and generic software engineering activities.

Grady argues that there are private and public uses for different types of process data. Individual software engineers might be sensitive to the use of metrics collected on an individual basis, so these data should be private to the individual and serve as an indicator for the individual only. Examples of private metrics include defect rates (by individual), defect rates (by component), and errors found during development.

The “private process data” philosophy aligns with the Personal Software Process approach proposed by Humphrey, who recognized that software process improvement can and should begin at the individual level. Private process data can serve as an important driver for improving one's software engineering approach.

Some process metrics are private to the software project team but public to all team members. Examples include defects reported for major software functions, errors found during technical reviews, and lines of code or function points per component or function. The team reviews these data to uncover indicators that can improve team performance.

Public metrics generally assimilate information that was originally private to individuals and teams. Project-level defect rates, effort, calendar times, and related data are collected and evaluated to uncover indicators that can improve organizational process performance.

Software process metrics can provide significant benefits as an organization works to improve its overall level of process maturity. However, these metrics can be misused, creating more problems than they solve. Grady suggests a "software metrics etiquette" that is appropriate for both managers and practitioners:

- Use common sense and organizational sensitivity when interpreting metrics data.
- Provide regular feedback to the individuals and teams who collect measures and metrics.
- Don’t use metrics to appraise individuals.
- Work with practitioners and teams to set clear goals and metrics that will be used to achieve them.
- Never use metrics to threaten individuals or teams.

- Metrics data that indicate a problem area should not be considered "negative." These data are merely indicators for process improvement.
- Don’t obsess on a single metric to the exclusion of other important metrics.

As an organization becomes more comfortable with the collection and use of process metrics, the derivation of simple indicators gives way to a more rigorous approach called statistical software process improvement (SSPI). SSPI uses software failure analysis to collect information about all errors and defects encountered as an application, system, or product is developed and used.

32.1.2 Project Metrics

Unlike software process metrics used for strategic purposes, software project measures are tactical. Project metrics and the indicators derived from them are used by a project manager and a software team to adapt project workflow and technical activities.

The first application of project metrics on most software projects occurs during estimation. Metrics collected from past projects are used as a basis for effort and time estimates for current software work. As a project proceeds, measures of effort and calendar time expended are compared to original estimates (and the project schedule). The project manager uses these data to monitor and control progress.

As technical work commences, other project metrics begin to have significance. Production rates represented in terms of models created, review hours, function points, and delivered source lines are measured. Errors uncovered during each software engineering task are also tracked. As the software evolves from requirements into design, technical metrics are collected to assess design quality and provide indicators that will influence the approach to code generation and testing.

The intent of project metrics is twofold:

1. To minimize the development schedule by making adjustments necessary to avoid delays and mitigate potential problems and risks.
2. To assess product quality on an ongoing basis and, when necessary, modify the technical approach to improve quality.

As quality improves, defects are minimized, and as the defect count decreases, the amount of rework required during the project is also reduced. This leads to a reduction in overall project cost.

32.2 SOFTWARE MEASUREMENT:

32.2.1 Size-Oriented Metrics

Size-oriented software metrics are derived by normalizing quality and/or productivity measures by considering the size of the software that has been produced. If a software organization maintains simple records, a table of size-oriented measures, such as the one shown in Figure 32.2, can be created. The table lists each software development project that has been completed over the past few years and corresponding measures for that project. Referring to the table entry (Figure 32.2) for project alpha: 12,100 lines of code were developed with 24 person-months of effort at a cost of $168,000.

Effort and cost recorded in the table represent all software engineering activities (analysis, design, code, and test), not just coding. Further information for project alpha indicates that 365 pages of documentation were developed, 134 errors were recorded before the software was released, and 29 defects were encountered after release to the customer within the first year of operation. Three people worked on the development of software for project alpha.

Size-oriented metrics are not universally accepted as the best way to measure the software process. Most of the controversy swirls around the use of lines of code as a key measure. Proponents of the LOC measure claim that LOC is an "artifact" of all software development projects that can be easily counted, that many existing software estimation models use LOC or KLOC as a key input, and that a large body of literature and data predicated on LOC already exists. On the other hand, opponents argue that LOC measures are programming language dependent, that when productivity is considered, they penalize well-designed but shorter programs; that they cannot easily accommodate nonprocedural languages; and that their use in estimation requires a level of detail that may be difficult to achieve (i.e., the planner must estimate the LOC to be produced long before analysis and design have been completed).

FIGURE 32.2
Size-oriented metrics

Project	LOC	Effort	$(000)	Pp. doc.	Errors	Defects	People
alpha	12,100	24	168	365	134	29	3
beta	27,200	62	440	1224	321	86	5
gamma	20,200	43	314	1050	256	64	6
•	•	•	•	•	•		
•	•	•	•	•	•		
•	•	•	•	•	•		

32.2.2 Function-Oriented Metrics

Function-oriented software metrics use a measure of the functionality delivered by the application as a normalization value. The most widely used function-oriented metric is the function point (FP). Computation of the function point is based on characteristics of the software's information domain and complexity. The mechanics of FP computation have been discussed in Chapter 30.

Size-oriented metrics are widely used, but debate about their validity and applicability continues.

The function point, like the LOC measure, is controversial. Proponents claim that FP is programming language–independent, making it ideal for applications using conventional and nonprocedural languages, and that it is based on data that are more likely to be known early in the evolution of a project, making FP more attractive as an estimation approach. Opponents claim that the method requires some "sleight of hand" in that computation is based on subjective rather than objective data, that counts of the information domain (and other dimensions) can be difficult to collect after the fact, and that FP has no direct physical meaning—it's just a number.

32.2.3 Reconciling LOC and FP Metrics

The relationship between lines of code and function points depends upon the programming language that is used to implement the software and the quality of the design. A number of studies have attempted to relate FP and LOC measures. The following table provides rough estimates of the average number of lines of code required to build one function point in various programming languages:

	LOC per Function Point			
Programming Language	**Avg.**	**Median**	**Low**	**High**
Ada	154	—	104	205
ASP	56	50	32	106
Assembler	337	315	91	694
C	148	107	22	704
C++	59	53	20	178
C#	58	59	51	704
COBOL	80	78	8	400
ColdFusion	68	56	52	105
DBase IV	52	—	—	—
Easytrieve+	33	34	25	41
Focus	43	42	32	56
FORTRAN	90	118	35	—
FoxPro	32	35	25	35
HTML	43	42	35	53
Informix	42	31	24	57
J2EE	57	50	50	67
Java	55	53	9	214
JavaScript	54	55	45	63
JSP	59	—	—	—
Lotus Notes	23	21	15	46

A review of these data indicates that one LOC of C++ provides approximately 2.4 times the "functionality" (on average) as one LOC of C. Furthermore, one LOC of Smalltalk provides at least four times the functionality of an LOC for a conventional programming language such as Ada, COBOL, or C. Using the information contained in the table, it is possible to "backfire" existing software to estimate the number of function points, once the total number of programming language statements are known.

LOC and FP measures are often used to derive productivity metrics. This invariably leads to a debate about the use of such data. Should the LOC/person-month (or FP/person-month) of one group be compared to similar data from another? Should managers appraise the performance of individuals by using these metrics? The answer to these questions is an emphatic no! The reason for this response is that many factors influence productivity, making for "apples and oranges" comparisons that can be easily misinterpreted.

Function points and LOC-based metrics have been found to be relatively accurate predictors of software development effort and cost. However, in order to use LOC and FP for estimation, an historical baseline of information must be established.

Within the context of process and project metrics, you should be concerned primarily with productivity and quality—measures of software development "output" as a function of effort and time applied and measures of the "fitness for use" of the work products that are produced. For process improvement and project planning purposes, your interest is historical. What was software development productivity on past projects? What was the quality of the software that was produced? How can past productivity and quality data be extrapolated to the present? How can it help us improve the process and plan new projects more accurately?

32.2.4 Object-Oriented Metrics

Conventional software project metrics (LOC or FP) can be used to estimate object-oriented software projects. However, these metrics do not provide enough granularity for the schedule and effort adjustments that are required as you iterate through an evolutionary or incremental process. Lorenz and Kidd [Lor94] suggest the following set of metrics for OO projects:

- **Number of scenario scripts**: A scenario script (analogous to a use case) is a detailed sequence of steps that describes the interaction between the user and the application. Each script is organized into triplets of the form {initiator, action, participant} where initiator is the object that requests some service (that initiates a message), action is the result of the request, and participant is the server object that satisfies the request. The number of scenario scripts is directly correlated to the size of the application and to the number of test cases that must be developed to exercise the system once it is constructed.

- **Number of key classes**: Key classes are the "highly independent components" [Lor94] that are defined early in object-oriented analysis (Chapter 10). Because key classes are central to the problem domain, the number of such classes is an indication of the amount of effort required to develop the software and also an indication of the potential amount of reuse to be applied during system development.
- **Number of support classes**: Support classes are required to implement the system but are not immediately related to the problem domain. Examples might be user interface (UI) classes, database access and manipulation classes, and computation classes. In addition, support classes can be developed for each of the key classes. Support classes are defined iteratively throughout an evolutionary process. The number of support classes is an indication of the amount of effort required to develop the software and also an indication of the potential amount of reuse to be applied during system development.
- **Average number of support classes per key class**: In general, key classes are known early in the project. Support classes are defined throughout. If the average number of support classes per key class were known for a given problem domain, estimating (based on total number of classes) would be greatly simplified.
- **Number of subsystems**. A subsystem is an aggregation of classes that support a function that is visible to the end user of a system. Once subsystems are identified, it is easier to lay out a reasonable schedule in which work on subsystems is partitioned among project staff.

  To be used effectively in an object-oriented software engineering environment, metrics similar to those noted above must be collected along with project measures such as effort expended, errors and defects uncovered, and models or documentation pages produced. As the database grows (after a number of projects have been completed), relationships between the object-oriented measures and project measures will provide metrics that can aid in project estimation.

32.2.5 Use Case-Oriented Metrics

Use cases are used widely as a method for describing customer-level or business domain requirements that imply software features and functions. It would seem reasonable to use the use case as a normalization measure similar to LOC or FP. Like FP, the use case is defined early in the software process, allowing it to be used for estimation before significant modeling and construction activities are initiated. Use cases describe (indirectly, at least) user-visible functions and features that are basic requirements for a system. The use case is independent of programming language. In addition, the number of use cases is directly proportional to the size of the application in LOC and to the number of test cases that will have to be designed to fully exercise the application.

Because use cases can be created at vastly different levels of abstraction, there is no standard "size" for a use case. Without a standard "measure" of what a use case is, its application as a normalization measure (e.g., effort expended per use case) is suspect.

Researchers have suggested use-case points (UCPs) as a mechanism for estimating project effort and other characteristics. The UCP is a function of the number of actors and transactions implied by the use-case models and is analogous to the FP in some ways. If you have further interest, see [Coh05], [Cle06], or [Col09].

32.2.6 WebApp Project Metrics

The objective of all WebApp projects is to deliver a combination of content and functionality to the end user. Measures and metrics used for traditional software engineering projects are difficult to translate directly to WebApps. Yet, it is possible to develop a database that allows access to internal productivity and quality measures derived over a number of projects. Among the measures that can be collected are the following:

- **Number of static Web pages**: These pages represent low relative complexity and generally require less effort to construct than dynamic pages. This measure provides an indication of the overall size of the application and the effort required to develop it.
- **Number of dynamic Web pages**: These pages represent higher relative complexity and require more effort to construct than static pages. This measure provides an indication of the overall size of the application and the effort required to develop it.
- **Number of internal page links**: This measure provides an indication of the degree of architectural coupling within the WebApp. As the number of page links increases, the effort expended on navigational design and construction also increases.
- **Number of persistent data objects**: As the number of persistent data objects (e.g., a database or data file) grows, the complexity of the WebApp also grows and the effort to implement it increases proportionally.
- **Number of external systems interfaced**: As the requirement for interfacing grows, system complexity and development effort also increase.
- **Number of static content objects**: These objects represent low relative complexity and generally require less effort to construct than dynamic pages.
- **Number of dynamic content objects**: These objects represent higher relative complexity and require more effort to construct than static pages.
- **Number of executable functions**: As the number of executable functions (e.g., a script or applet) increases, modelling and construction effort also increase.

Briefly explain various decomposition techniques used during software project estimations.

Software project estimation is a form of problem solving, and in most cases, the problem to be solved (i.e., developing a cost and effort estimate for a software project) is too complex to be considered in one piece. For this reason, you should decompose the problem, recharacterizing it as a set of smaller (and hopefully, more manageable) problems. In Chapter 31, the decomposition approach was discussed from two different

points of view: decomposition of the problem and decomposition of the process. Estimation uses one or both forms of partitioning. But before an estimate can be made, you must understand the scope of the software to be built and generate an estimate of its “size.”

33.6.1 Software Sizing

The accuracy of a software project estimate relies on:

1. Properly estimating the product size.
2. Translating the size estimate into effort, time, and cost using past project metrics.
3. Reflecting the software team's abilities in the project plan.
4. Stability of product requirements and supporting environment.

Software sizing, a key challenge in project planning, refers to the quantifiable outcome of the project. It can be measured directly in lines of code (LOC) or indirectly in function points (FP). Size estimates consider the project type, application domain, functionality delivered, number of components, and modifications needed for existing components.

Putnam and Myers [Put92] recommend combining optimistic, most likely, and pessimistic size values to create a three-point estimate, as detailed in Section 33.6.2.

33.6.2 Problem-Based Estimation

In Chapter 32, lines of code (LOC) and function points (FP) were described as measures for computing productivity metrics. These metrics are used in software project estimation to:

1. Size each software element.
2. Use past project data to develop cost and effort projections.

Both LOC and FP estimation techniques begin with a defined software scope, decomposing it into smaller problem functions for individual estimation. LOC or FP values are estimated for each function. Alternatively, other components like classes, objects, changes, or business processes may be used for sizing.

Baseline productivity metrics (e.g., LOC/pm or FP/pm) are then applied to the estimation variable to derive cost or effort for each function. These estimates are combined to produce an overall project estimate.

It's important to note that productivity metrics can vary widely within an organization. Therefore, averages should be computed by project domain, considering team size, application area, complexity, and other parameters. These domain-specific averages should be used for new project estimates.

The level of detail required for decomposition differs between LOC and FP:

- **LOC Estimation**: Requires detailed decomposition. The more detailed the partitioning, the more accurate the LOC estimates.
- **FP Estimation**: Focuses on information domain characteristics (inputs, outputs, data files, inquiries, external interfaces) and complexity adjustments.

For both LOC and FP, it's useful to estimate a range of values (optimistic, most likely, and pessimistic) for each function or information domain value. This provides an indication of the degree of uncertainty.

The expected value for the estimation variable (size) S can be computed as:

S=sopt+4sm+spess6S = \frac{s_{opt} + 4s_m + s_{pess}}{6}S=6sopt+4sm+spess

This formula gives the most weight to the "most likely" estimate and assumes a beta probability distribution.

After determining the expected value, historical LOC or FP productivity data are applied. Cross-checking with another estimation technique and using common sense and experience is essential to ensure accuracy.

33.6.5 Process-Based Estimation

The most common technique for estimating a project is based on the process that will be used. The process is broken down into a set of activities, actions, and tasks, and the effort required for each is estimated.

Process-based estimation starts with defining software functions from the project scope. Each function involves a series of framework activities, which can be represented in a table. For each software function, you estimate the effort (e.g., person-months) needed for each process activity. Average labor rates (cost per unit effort) are then applied to these estimates.

If process-based estimation is done independently of LOC or FP estimation, you end up with two or three estimates for cost and effort, which can be compared and reconciled. If the estimates agree, they are likely reliable. If not, further investigation is necessary.

For finer granularity, break down major tasks like analysis into smaller tasks and estimate each separately.

33.6.7 Estimation with Use Cases

Use cases provide insight into software scope and requirements, and can be used to estimate the projected "size" of a software project. However, use cases vary in format and abstraction, making estimation challenging. Despite this, it is possible to compute use case points (UCPs) similarly to function points.

To compute UCPs, consider the following characteristics:

- Number and complexity of use cases.
- Number and complexity of actors.
- Nonfunctional requirements (e.g., portability, performance).
- Development environment (e.g., programming language, team motivation).

Steps to Compute UCPs:

1. **Assess Use Case Complexity:**
    - Simple use cases: Simple UI, single database, ≤3 transactions, ≤5 classes.
    - Average use cases: More complex UI, 2-3 databases, 4-7 transactions, 5-10 classes.

- Complex use cases: Complex UI, multiple databases, ≥8 transactions, ≥11 classes.

Weight each type by factors of 5, 10, and 15 respectively. Sum these weighted counts to get the total unadjusted use case weight (UUCW).

2. **Assess Actor Complexity:**
   - Simple actors: Automatons communicating through an API.
   - Average actors: Automatons communicating through a protocol or data store.
   - Complex actors: Humans communicating through a GUI or other interface.

   Weight each type by factors of 1, 2, and 3 respectively. Sum these weighted counts to get the total unadjusted actor weight (UAW).

3. **Adjust for Complexity Factors:**
   - Technical Complexity Factors (TCFs): Thirteen factors contribute to the final TCF.
   - Environment Complexity Factors (ECFs): Eight factors contribute to the final ECF.
4. **Calculate UCP:** UCP=(UUCW+UAW)×TCF×ECF\text{UCP} = (\text{UUCW} + \text{UAW}) \times \text{TCF} \times \text{ECF}UCP=(UUCW+UAW)×TCF×ECF

33.6.9 Reconciling Estimates

The estimation techniques discussed result in multiple estimates that must be reconciled to produce a single estimate of effort, project duration, or cost. For example, the total estimated effort for CAD software ranges from 46 person-months (process-based estimation) to 68 person-months (use-case estimation), with an average of 56 person-months. This variation is approximately 18% below and 21% above the average estimate.

Handling Poor Agreement Between Estimates

When estimates differ significantly, reevaluate the information used to make them. Divergent estimates often arise from:

1. **Scope Misunderstanding**: The project scope might not be adequately understood or has been misinterpreted.
2. **Inappropriate Productivity Data**: The data used for problem-based estimation might be inappropriate, obsolete, or misapplied.

Determine the cause of the divergence and reconcile the estimates accordingly.

Key Points

- Ensure the project scope is clearly understood and communicated.
- Use up-to-date and relevant productivity data.
- Cross-check estimates with different techniques for reliability.
- Incorporate expert intuition and experience in the final reconciliation.

Describe Empirical estimation models used during estimation of software projects.

An estimation model for computer software uses empirically derived formulas to predict effort as a function of LOC or FP.9 Values for LOC or FP are estimated using the approach described in Sections 33.6.3 and 33.6.4. But instead of using the tables described in those sections, the resultant values for LOC or FP are plugged into the estimation model. The empirical data that support most estimation models are derived from a limited sample of projects. For this reason, no estimation model is appropriate for all classes of software and in all development environments. Therefore, you should use the results obtained from such models judiciously.

An estimation model should be calibrated to reflect local conditions. The model should be tested by applying data collected from completed projects, plugging the data into the model, and then comparing actual to predicted results. If agreement is poor, the model must be tuned and retested before it can be used.

33.7.1 The Structure of Estimation Models

A typical estimation model is derived using regression analysis on data collected from past software projects. The overall structure of such models takes the form [Mat94]

$$E = A + B \times (e_v)^C$$

where A, B, and C are empirically derived constants, E is effort in person-months, and ev is the estimation variable (either LOC or FP). In addition to the relationship noted in Equation (33.3), the majority of estimation models have some form of project adjustment component that enables E to be adjusted by other project characteristics (e.g., problem complexity, staff experience, development environment). A quick examination of any empirically derived model indicates that it must be calibrated for local needs.

33.7.2 The COCOMO II Model

In his classic book on software engineering economics, Barry Boehm [Boe81] introduced a hierarchy of software estimation models bearing the name COCOMO, for Constructive Cost Model. The original COCOMO model became one of the most widely used and discussed software cost estimation models in the industry. It has evolved into a more comprehensive estimation model, called COCOMO II [Boe00].

Like its predecessor, COCOMO II is actually a hierarchy of estimation models that address different "stages" of the software process. Like all estimation models for software, the COCOMO II models require sizing information. Three different sizing options are available as part of the model hierarchy: object points, 10 function points, and lines of source code.

33.7.3 The Software Equation

The software equation is a dynamic multivariable model derived from productivity data of over 4,000 contemporary software projects. It estimates effort based on the distribution of effort over the life of a software development project. The model is represented as:

$$E = \frac{LOC^{0.333}}{P^3 \cdot t^4}$$

Where:

- **E** = Effort in person-months or person-years
- **t** = Project duration in months or years
- **B** = Special skills factor
- **P** = Productivity parameter, reflecting factors like process maturity, management practices, software engineering practices, programming languages used, software environment, team skills and experience, and application complexity

Typical values for **P**:

- Real-time embedded software: $P = 2,000$
- Telecommunication and systems software: $P = 10,000$
- Business systems applications: $P = 28,000$

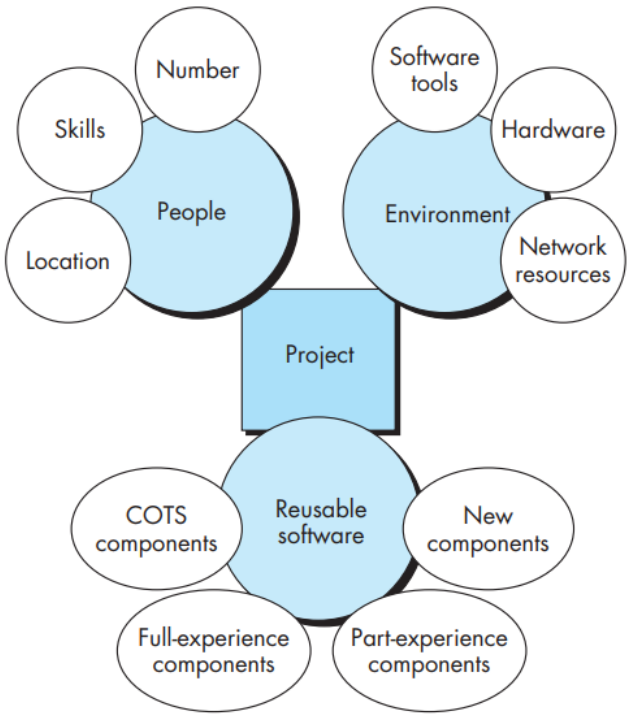
Simplified Estimation Equations

Putnam and Myers suggest simplified equations for estimating development time and effort:

1. **Minimum Development Time**:
   $t_{min} = 8.14 \cdot \left(\frac{LOC}{P^{0.43}}\right)$
   - Applicable for $t_{min} > 6$ months
   - Example: For CAD software with $LOC = 33,200$ and $P = 12,000$,
     $t_{min} = 8.14 \cdot \left(\frac{33,200}{12,000^{0.43}}\right) = 12.6$ calendar months
2. **Effort**:
   $E = 180 \cdot B \cdot t^3$
   - Applicable for $E \geq 20$ person-months (where t is in years)
   - Example: For $t = 1.05$ years,
     $E = 180 \cdot 0.28 \cdot (1.05)^3 = 58$ person-months

List and explain the three major categories of software engineering resources.

FIGURE 33.1
Project resources



The second planning task is estimation of the resources required to accomplish the software development effort. Figure 33.1depicts the three major categories of software engineering resources—people, reusable software components, and the development environment (hardware and software tools). Each resource is specified with four characteristics: description of the resource, a statement of availability, time when the resource will be required, and duration of time that the resource will be applied. The last two characteristics can be viewed as a time window. Availability of the resource for a specified window must be established at the earliest practical time.

33.4.1 Human Resources The planner begins by evaluating software scope and selecting the skills required to complete development. Both organizational position (e.g., manager, senior software engineer) and specialty (e.g., telecommunications, database, client-server) are specified. For relatively small projects (a few person-months), a single individual may perform all software engineering tasks, consulting with specialists as required. For larger projects, the software team may be geographically dispersed across a number of different locations. Hence, the location of each human resource is specified. The number of

people required for a software project can be determined only after an estimate of development effort (e.g., person-months) is made. Techniques for estimating effort are discussed later in this chapter.

33.4.2 Reusable Software Resources

Component-based software engineering (CBSE)4 emphasizes reusability—that is, the creation and reuse of software building blocks. Such building blocks, often called components, must be catalogued for easy reference, standardized for easy application, and validated for easy integration. Bennatan [Ben00] suggests

four software resource categories that should be considered as planning proceeds:

off-the-shelf components (existing software that can be acquired from a third party or from a past project), full-experience components (existing specifications, designs, code, or test data developed for past projects that are similar to the software to be built for the current project),

partial-experience components (existing specifications, designs, code, or test data developed for past projects that are related to the software to be built for the current project but will require substantial modification), and

new components (components built by the software team specifically for the needs of the current project). Ironically, reusable software components are often neglected during planning, only to become a paramount concern during the development phase of the software process. It is better to specify software resource requirements early. In this way technical evaluation of the alternatives can be conducted and timely acquisition can occur.

33.4.3 Environmental Resources

The environment that supports a software project, often called the software engineering environment (SEE), incorporates hardware and software. Hardware provides a platform that supports the tools (software) required to produce the work products that are an outcome of good software engineering practice. 5 Because most software organizations have multiple constituencies that require access to the SEE, you must prescribe the time window required for hardware and software and verify that these resources will be available. When a computer-based system (incorporating specialized hardware and software) is to be engineered, the software team may require access to hardware elements being developed by other engineering teams. For example, software for a robotic device used within a manufacturing cell may require a specific robot (e.g., a robotic welder) as part of the validation test step; a software project for advanced page layout may need a high-speed digital printing system at some point during development. Each hardware element must be specified as part of planning.