UNIT IV SOFTWARE QUALITY AND PROJECT MANAGEMENT

Elements of SQA – SQA Tasks, Goals, Metrics – Statistical SQA – Software Reliability – ISO 9000 Quality Standards – SQA Plan – Project management spectrum – People – Product- Process – Project – W^5HH Principle – Critical Practices.

4.1 ELEMENTS OF SOFTWARE QUALITY ASSURANCE (SQA)

Software quality assurance encompasses a broad range of concerns and activities that focus on the management of software quality.

Standards: The IEEE, ISO, and other standards organizations have produced a broad array of software engineering standards and related documents. Standards may be adopted voluntarily by a software engineering organization or imposed by the customer or other stakeholders. The job of SQA is to ensure that standards that have been adopted are followed and that all work products conform to them.

Reviews and audits: Technical reviews are a quality control activity performed by software engineers for software engineers. Their intent is to uncover errors. Audits are a type of review performed by SQA personnel with the intent of ensuring that quality guidelines are being followed for software engineering work. For example, an audit of the review process might be conducted to ensure that reviews are being performed in a manner that will lead to the highest likelihood of uncovering errors.

Testing: Software testing is a quality control function that has one primary goal—to find errors. The job of SQA is to ensure that testing is properly planned and efficiently conducted so that it has the highest likelihood of achieving its primary goal.

Error/defect collection and analysis: The only way to improve is to measure how you're doing. SQA collects and analyzes error and defect data to better understand how errors are introduced and what software engineering activities are best suited to eliminating them.

Change management: Change is one of the most disruptive aspects of any software project. If it is not properly managed, change can lead to confusion, and confusion almost always leads to poor quality. SQA ensures that adequate change management practices have been instituted.

Education: Every software organization wants to improve its software engineering practices. A key contributor to improvement is education of software engineers, their managers, and other stakeholders. The SQA organization takes the lead in software process improvement and is a key proponent and sponsor of educational programs.

Vendor management: Three categories of software are acquired from external software vendors—shrink-wrapped packages (e.g., Microsoft Office), a tailored shell [Hor03] that provides a basic skeletal structure that is custom tailored to the needs of a purchaser, and contracted software that is custom designed and constructed from specifications provided by the customer organization. The job of the SQA organization is to ensure that high-quality software results by suggesting specific quality practices

that the vendor should follow (when possible), and incorporating quality mandates as part of any contract with an external vendor.

Security management: With the increase in cyber crime and new government regulations regarding privacy, every software organization should institute policies that protect data at all levels, establish firewall protection for WebApps, and ensure that software has not been tampered with internally. SQA ensures that appropriate process and technology are used to achieve software security.

Safety: Because software is almost always a pivotal component of human-rated systems (e.g., automotive or aircraft applications), the impact of hidden defects can be catastrophic. SQA may be responsible for assessing the impact of software failure and for initiating those steps required to reduce risk.

Risk management: Although the analysis and mitigation of risk is the concern of software engineers, the SQA organization ensures that risk management activities are properly conducted and that risk-related
contingency plans have been established.

In addition to each of these concerns and activities, SQA works to ensure that soft-
ware support activities (e.g., maintenance, help lines, documentation, and manuals)
are conducted or produced with quality as a dominant concern.

4.2 SQA TASKS, GOALS AND METRICS

Software quality assurance is composed of a variety of tasks associated with two different constituencies—the software engineers who do technical work and an SQA group that has responsibility for quality assurance planning, oversight, record keeping, analysis, and reporting.

Software engineers address quality (and perform quality control activities) by applying solid technical methods and measures, conducting technical reviews, and performing well-planned software testing.

1. SQA Tasks:

The charter of the SQA group is to assist the software team in achieving a high-quality end product. The Software Engineering Institute recommends a set of SQA actions that address quality assurance planning, oversight, record keeping, analysis, and reporting. These actions are performed (or facilitated) by an independent SQA group that:

Prepares an SQA plan for a project. The plan is developed as part of project planning and is reviewed by all stakeholders. Quality assurance actions performed by the software engineering team and the SQA group are governed by the plan. The plan identifies evaluations to be performed, audits and reviews to be conducted, standards that are applicable to the project, procedures for error reporting and tracking,

work products that are produced by the SQA group, and feedback that will be provided to the software team.

Participates in the development of the project's software process description. The software team selects a process for the work to be performed. The SQA group reviews the process description for compliance with organizational policy, internal software standards, externally imposed standards (e.g., ISO-9001), and other parts of the software project plan.

Reviews software engineering activities to verify compliance with the defined software process. The SQA group identifies, documents, and tracks deviations from the process and verifies that corrections have been made.

Audits designated software work products to verify compliance with those defined as part of the software process. The SQA group reviews selected work products; identifies, documents, and tracks deviations; verifies that corrections have been made; and periodically reports the results of its work to the project manager.

Ensures that deviations in software work and work products are documented and handled according to a documented procedure. Deviations may be encountered in the project plan, process description, applicable standards, or software engineering work products.

Records any noncompliance and reports to senior management. Noncompliance items are tracked until they are resolved.

In addition to these actions, the SQA group coordinates the control and management of change and helps to collect and analyze software metrics.

2. Goals, Attributes, and Metrics:

The SQA actions described in the preceding section are performed to achieve a set of pragmatic goals:

Requirements quality. The correctness, completeness, and consistency of the requirements model will have a strong influence on the quality of all work products that follow. SQA must ensure that the software team has properly reviewed the requirements model to achieve a high level of quality.

Design quality. Every element of the design model should be assessed by the software team to ensure that it exhibits high quality and that the design itself conforms to requirements. SQA looks for attributes of the design that are indicators of quality.

Code quality. Source code and related work products (e.g., other descriptive information) must conform to local coding standards and exhibit characteristics that will facilitate maintainability. SQA should isolate those attributes that allow a reasonable analysis of the quality of code.

Quality control effectiveness: A software team should apply limited resources in a way that has the

highest likelihood of achieving a high-quality result. SQA analyzes the allocation of resources for reviews and testing to assess whether they are being allocated in the most effective manner.

Goal	Attribute	Metric
Requirement quality	Ambiguity	Number of ambiguous modifiers (e.g., many, large, human-friendly)
	Completeness	Number of TBA, TBD
	Understandability	Number of sections/subsections
	Volatility	Number of changes per requirement
		Time (by activity) when change is requested
	Traceability	Number of requirements not traceable to design/code
	Model clarity	Number of UML models
		Number of descriptive pages per model
		Number of UML errors
Design quality	Architectural integrity	Existence of architectural model
	Component completeness	Number of components that trace to architectural model
		Complexity of procedural design
	Interface complexity	Average number of pick to get to a typical function or content
		Layout appropriateness
	Patterns	Number of patterns used
Code quality	Complexity	Cyclomatic complexity
	Maintainability	Design factors
	Understandability	Percent internal comments
		Variable naming conventions
	Reusability	Percent reused components
	Documentation	Readability index
QC effectiveness	Resource allocation	Staff hour percentage per activity
	Completion rate	Actual vs. budgeted completion time
	Review effectiveness	See review metrics
	Testing effectiveness	Number of errors found and criticality
		Effort required to correct an error
		Origin of error

4.3 STATISTICAL SOFTWARE QUALITY ASSURANCE

Statistical quality assurance reflects a growing trend throughout industry to become more quantitative about quality. For software, statistical quality assurance implies the following steps:

1.Information about software defects is collected and categorized.

2.An attempt is made to trace each defect to its underlying cause (e.g., non-conformance to specifications, design error, violation of standards, poor communication with the customer).

3. Using the Pareto principle (80 percent of the defects can be traced to 20 percent of all possible causes),

isolate the 20 percent (the "vital few").

4. Once the vital few causes have been identified, move to correct the problems that have caused the defects.

This relatively simple concept represents an important step towards the creation o: an adaptive software engineering process in which changes are made to improve those elements of the process that introduce error.

To illustrate this, assume that a software engineering organization collects information on defects for a period of one year. Some of the defects are uncovered as software is being developed. Others are encountered after the software has been released to its end-users.

Although hundreds of different errors are uncovered, all can be tracked to one (or more) of the following causes:

•Incomplete or erroneous specifications (IBS)

•Misinterpretation of customer communication (MCC)

•Intentional deviation from specifications (IDS)

•Violation of programming standards (VPS)

•Error in data representation (EDR)

•Inconsistent component interface (ICI)

•Error in design logic (EDL)

•Incomplete or erroneous testing (IET)

•Inaccurate or incomplete documentation (IID)

•Error in programming language translation of design (PLT)

•Ambiguous or inconsistent human/computer interface (HCI)

•Miscellaneous (MIS)

To apply statistical SQA, Table 1.1 is built. The table indicates that IES, MCC, and EDR are the vital few causes that account for 53 percent of all errors. It should be noted, however, that IES, EDR, PLT, and EDL would be selected as the vital few causes if only serious errors are considered. Once the vital few causes are determined, the software engineering organization can begin corrective action.

For example, to correct MCC, the software developer might implement facilitated application specification techniques to improve the quality of customer communication and specifications.

To improve EDR, the developer might acquire CASE tools for data modeling and perform more stringent data design reviews. It is important to note that corrective action focuses primarily on the vital few. As the vital few causes are corrected, new candidates pop to the top of the stack.

Statistical quality assurance techniques for software have been shown to provide substantial quality improvement. In some cases, software organizations have achieved a 50 percent reduction per year in defects after applying these techniques.

In conjunction with the collection of defect information, software developers can calculate an error index (El) for each major step in the software process. After analysis, design, coding, testing, and release, the following data are gathered:

E i = The total number of errors uncovered during the ith step in the software engineering process.

Table 4.1: Data Collection for Statistical SQA

	Total		Serious		Moderate		Minor	
Error	**No.**	**%**	**No.**	**%**	**No.**	**%**	**No.**	**%**
IES	205	22%	34	27%	68	18%	103	24%
MCC	156	17%	12	9%	68	18%	76	17%
IDS	48	5%	1	1%	24	6%	23	5%
VPS	25	3%	0	0%	15	4%	10	2%
EDR	130	14%	26	20%	68	18%	36	8%
ICI	58	6%	9	7%	18	5%	31	7%
EDL	45	5%	14	11%	12	3%	19	4%
IET	95	10%	12	9%	35	9%	48	11%
IID	36	4%	2	2%	20	5%	14	3%
PLT	60	6%	15	12%	19	5%	26	6%
HCI	28	3%	3	2%	17	4%	8	2%
MIS	56	6%	0	0%	15	4%	41	9%
Totals	942	100%	128	100%	379	100%	435	100%

S_i = the number of serious errors

M_i = the number of moderate errors

T_i = the number of minor errors

PS = size of the product (LOG, design statements, pages of documentation) at the ith step

W_s, W_m, W_t = weighting factors for serious, moderate, and trivial errors, where recommended values are $W_s = 10$, $W_m = 3$, $W_t = 1$. The weighting factors for each phase should become larger as development progresses. This rewards an organization that finds errors early.

At each step in the software process, a phase index, PI_i, is computed:

$$Pl_i = W_s (S_i/E_i) + W_m (M_i/E_i) + W_t (T_i/E_i)$$

The error index is computed by calculating the cumulative effect on each PI_i, weighting errors encountered later in the software engineering process more heavily than those encountered earlier:

$$EI = \Sigma(I \times PI_i)/PS = (PI_i + 2PI_2 + 3PI_3 + \ldots iPI_i)/PS$$

The error index can be used in conjunction with information collected.in Table 4.1.To develop an overall indication of improvement in software quality.

4.4 SOFTWARE RELIABILITY

Software reliability is defined as the probability of failure-free operation of a software system for a specified time in a specified environment.

Learning Objectives:

• To differentiate the failure and faults.
• To highlight the importance of execution and calendar time
• To understand Time interval between failures.
• To understand on the user perception of reliability.

Definitions Of Software Reliability:

Software reliability is defined as the probability of failure-free operation of a software system for a specified time in a specified environment. The key elements of the definition include probability of failure-free operation, length of time of failure-free operation and the given execution environment. Failure intensity is a measure of the reliability of a software system operating in a given environment. Example: An air traffic control system fails once in two years.

Factors Influencing Software Reliability

- A user's perception of the reliability of a software depends upon two categories of information.
    - The number of faults present in the software.
    - The way users operate the system. This is known as the *operational profile*.

- The fault count in a system is influenced by the following.
    - Size and complexity of code.
    - Characteristics of the development process used.
    - Education, experience, and training of development personnel.
    - Operational environment.

Applications of Software Reliability:

The applications of software reliability includes

- **Comparison of software engineering technologies.**
    - What is the cost of adopting a technology?
    - What is the return from the technology — in terms of cost and quality?
- **Measuring the progress of system testing –**The failure intensity measure tells us about the present quality of the system: high intensity means more tests are to be performed.
- **Controlling the system in operation –**The amount of change to a software for maintenance affects its reliability.
- **Better insight into software development processes** – Quantification of quality gives us a better insight into the development processes.

Functional And Non-Functional Requirements:

System functional requirements may specify error checking, recovery features, and system failure protection. System reliability and availability are specified as part of the non-functional requirements for the system.

System Reliability Specification:

- Hardware reliability focuses on the probability a hardware component fails.
- Software reliability focuses on the probability a software component will produce an incorrect output.
- The software does not wear out and it can continue to operate after a bad result.
- Operator reliability focuses on the probability when a system user makes an error.

Failure Probabilities:

If there are two independent components in a system and the operation of the system depends on them both then, P(S) = P (A) + P (B)

If the components are replicated then the probability of failure is P(S) = P (A) n which means that all components fail at once.

Functional Reliability Requirements:

- The system will check all operator inputs to see that they fall within their required ranges.
- The system will check all disks for bad blocks each time it is booted.
- The system must be implemented in using a standard implementation of Ada.

Non-Functional Reliability Specification:

The required level of reliability must be expressed quantitatively. Reliability is a dynamic system attribute. Source code reliability specifications are meaningless (e.g. N faults/1000 LOC). An appropriate metric should be chosen to specify the overall system reliability.

Hardware Reliability Metrics:

Hardware metrics are not suitable for software since its metrics are based on notion of component failure. Software failures are often design failures. Often the system is available after the failure has occurred. Hardware components can wear out.

Software Reliability Metrics:

Reliability metrics are units of measure for system reliability. System reliability is measured by counting the number of operational failures and relating these to demands made on the system at the time of failure. A long-term measurement program is required to assess the reliability of critical systems.

Probability Of Failure On Demand:

The probability system will fail when a service request is made. It is useful when requests are made on an intermittent or infrequent basis. It is appropriate for protection systems where service requests may be rare and consequences can be serious if service is not delivered. It is relevant for many safety-critical systems with exception handlers.

Rate Of Fault Occurrence:

Rate of fault occurrence reflects upon the rate of failure in the system. It is useful when system has to process a large number of similar requests that are relatively frequent. It is relevant for operating systems and transaction processing systems.

Reliability Metrics:

- ***Probability of Failure on Demand (PoFoD)***
    - PoFoD = 0.001.
    - For one in every 1000 requests the service fails per time unit.
- ***Rate of Fault Occurrence (RoCoF)***
    - RoCoF = 0.02.
    - Two failures for each 100 operational time units of operation.
- ***Mean Time to Failure (MTTF)***
    - The average time between observed failures (aka MTBF)
    - It measures time between observable system failures.

    - For stable systems MTTF = 1/RoCoF.
    - It is relevant for systems when individual transactions take lots of processing time (e.g. CAD or WP systems).
- ***Availability = MTBF / (MTBF+MTTR)***
    - MTBF = Mean Time Between Failure
    - MTTR = Mean Time to Repair
- ***Reliability = MTBF / (1+MTBF)***

Time Units:

Time units include:

- **Raw Execution Time** which is employed in non-stop system.
- **Calendar Time** is employed when the system has regular usage patterns.
- **Number of Transactions** is employed for demand type transaction systems.

Availability:

Availability measures the fraction of time system is really available for use. It takes repair and restart times into account. It is relevant for non-stop continuously running systems (e.g. traffic signal).

Failure Consequences – Study 1

Reliability does not take consequences into account. Transient faults have no real consequences but other faults might cause data loss or corruption. Hence it may be worthwhile to identify different classes of failure, and use different metrics for each.

Failure Consequences – Study 2

When specifying reliability both the number of failures and the consequences of each matter. Failures with serious consequences are more damaging than those where repair and recovery is straightforward. In some cases, different reliability specifications may be defined for different failure types.

Failure Classification:

Failure can be classified as the following

- ***Transient*** – only occurs with certain inputs.
- ***Permanent*** – occurs on all
- ***Recoverable*** – system can recover without operator help.
- ***Unrecoverable –*** operator has to help.
- ***Non-corrupting*** – failure does not corrupt system state or d
- ***Corrupting*** – system state or data are altered.

Building Reliability Specification:

The building of reliability specification involves consequences analysis of possible system failures for each sub-system. From system failure analysis, partition the failure into appropriate classes. For each class send out the appropriate reliability metric.

Specification Validation:

It is impossible to empirically validate high reliability specifications. No database corruption really means PoFoD class < 1 in 200 million. If each transaction takes 1 second to verify, simulation of one day's transactions takes 3.5 days.

4.5 ISO 9000 QUALITY STANDARDS

A quality assurance system may be defined as the organizational structure, responsibilities, procedures, processes, and resources for implementing quality management. Quality assurance systems are created to help organizations ensure their products and services satisfy customer expectations by meeting their specifications. These systems cover a wide variety of activities encompassing a product's entire life cycle including planning, controlling, measuring, testing and reporting, and improving quality levels throughout the development and manufacturing process. ISO 9000 describes quality assurance elements in generic terms that can be applied to any business regardless of the products or services offered.

To become registered to one of the quality assurance system models contained in ISO 9000, a company's quality system and operations are scrutinized by third-party auditors for compliance to the standard and for effective operation. Upon successful registration, a company is issued a certificate from a registration body represented by the auditors. Semiannual surveillance audits ensure continued compliance to the standard.

The requirements delineated by ISO 9001:2000 address topics such as management responsibility, quality system, contract review, design control, document and data control, product identification and traceability, process control, inspection and testing, corrective and preventive action, control of quality records, internal quality audits, training, servicing, and statistical techniques. In order for a software organization to become registered to ISO 9001:2000, it must establish policies and procedures to address each of the requirements just noted (and others) and then be able to demonstrate that these policies and procedures are being followed.

4.6 SQA PLAN

The SQA Plan provides a road map for instituting software quality assurance.
Developed by the SQA group (or by the software team if an SQA group does not exist), the plan serves as atemplate for SQA activities that are instituted for each software project.
The standard recommends a structure that identifies:

1. The purpose and scope of the plan.
2. A description of all software engineering work products (e.g., models, documents, source code) that fallwithin the purview of SQA.
3. All applicable standards and practices that are applied during the software process.
4. SQA actions and tasks (including reviews and audits) and their placement throughout the softwareprocess.
5. The tools and methods that support SQA actions and tasks.

6. Software configuration management procedures.
7. Methods for assembling, safeguarding, and maintaining all SQA-related records.
8. Organizational roles and responsibilities relative to product quality.

4.7 PROJECT MANAGEMENT SPECTRUM

For properly building a product, there's a very important concept that we all should know in software project planning while developing a product. There are 4 critical components in software project planning which are known as the **4P's** namely:

- Product
- Process
- People
- Project

These components play a very important role in your project that can help your team meet its goals and objective. Now, Let's dive into each of them a little in detail to get a better understanding:

4.7.1 People

The most important component of a product and its successful implementation is human resources. In building a proper product, a well-managed team with clear-cut roles defined for each person/team will lead to the success of the product. We need to have a good team in order to save our time, cost, and effort. Some assigned roles in software project planning are **project manager, team leaders, stakeholders, analysts,** and other **IT professionals**. Managing people successfully is a tricky process which a good project manager can do.

In a study published by the IEEE, the engineering vice presidents of three major technology companies were asked what was the most important contributor to a successful software project. They answered in the following way:

VP 1: I guess if you had to pick one thing out that is most important in our environment, I'd say it's not the tools that we use, it's the people.

VP 2: The most important ingredient that was successful on this project was having smart people... very little else matters in my opinion.... The most important thing you do for a project is selecting the staff.... The success of the software development organization is very, very much associated with the ability to recruit good people.

VP 3: The only rule I have in management is to ensure I have good people-real good people and that I grow good people and that I provide an environment in which good people can produce.

4.7.1.1 The Stakeholders

The software process (and every software project) is populated by stakeholders who can be categorized into one of five constituencies:

1. Senior managers who define the business issues that often have a significant influence on the project.
2. Project (technical) managers who must plan, motivate, organize, and control the practitioners who do software work.
3. Practitioners who deliver the technical skills that are necessary to engineer a product or application.
4. Customers who specify the requirements for the software to be engineered and other stakeholders who have a peripheral interest in the outcome.

5. End users who interact with the software once it is released for production use. Every software project is populated by people who fall within this taxonomy. To be effective, the project team must be organized in a way that maximizes each person's skills and abilities. And that's the job of the team leader.

4.7.1.2 Team Leaders

Project management is a people intensive activity, and for this reason, competent practitioners often make poor team leaders. They simply don't have the right mix of people skills. And yet, as Edgemon states: "Unfortunately and all too frequently it seems, individuals just fall into a project manager role and become accidental project managers".

Jerry Weinberg suggests an MOI model of leadership:

Motivation:The ability to encourage (by "push or pull") technical people to produce to their best ability.

Organization:

The ability to mold existing processes (or invent new ones) that will enable the initial concept to be translated into a final product.

Ideas or innovation:

The ability to encourage people to create and feel creative even when they must work within bounds established for a particular software product or application.

Another view of the characteristics that define an effective project manager emphasizes four key traits:

Problem solving:

An effective software project manager can diagnose the technical and organizational issues that are most relevant, systematically structure a solution or properly motivate other practitioners to develop the solution, apply lessons learned from past projects to new situations, and remain flexible enough to change direction if initial attempts at problem solution are fruitless.

Managerial identity:

A good project manager must take charge of the project. She must have the confidence to assume control when necessary and the assurance to allow good technical people to follow their instincts. Achievement. A competent manager must reward initiative and accomplishment to optimize the productivity of a project team. She must demonstrate through her own actions that controlled risk taking will not be punished.

Influence and team building.:

An effective project manager must be able to "read" people; she must be able to understand verbal and nonverbal signals and react to the needs of the people sending these signals. The manager must remain under control in high-stress situations.

24.2.3 The Software Team

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 over- all problem difficulty. Mantei describes seven project factors that should be considered when planning the structure of software engineering teams:

- Difficulty of the problem to be solved
- "Size" of the resultant program(s) in lines of code or function points
- Time that the team will stay together (team lifetime)
- Degree to which the problem can be modularized
- Required quality and reliability of the system to be built
- Rigidity of the delivery date
- Degree of sociability (communication) required for the project

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.

As a counterpoint to the chief programmer team structure, Constantine's random paradigm [Con93] suggests a software team with creative independence whose approach to work might best be termed innovative anarchy. Although the free-spirited approach to software work has appeal, channeling creative energy into a high- performance team must be a central goal of a software engineering organization. To achieve a high-performance team:

- Team members must have trust in one another.
- The distribution of skills must be appropriate to the problem.
- Mavericks may have to be excluded from the team, if team cohesiveness is to be maintained.

4.7.1.4 Agile Teams

To review, the agile 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.

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.

If the people on the project are good enough, they can use almost any process and accomplish their assignment. If they are not good enough, no process will repair their inadequacy-"people trump process" is one way to say this. However, lack of user and executive support can kill a project-"politics trump people." Inadequate support can keep even good people from accomplishing the job.

To make effective use of the competencies of each team member and to foster effective collaboration through a software project, agile teams are self-organizing.

Many agile process models (e.g., Scrum) give the agile team significant autonomy to make the project management and technical decisions required to get the job done. Planning is kept to a minimum, and the team is allowed to select its own approach (e.g., process, methods, tools), constrained only by business requirements and organizational standards. As the project proceeds, the team self-organizes to focus individual competency in a way that is most beneficial to the project at a given point in time. To accomplish this, an agile team might conduct daily team meetings to coordinate and synchronize the work that must be accomplished for that day.

Based on information obtained during these meetings, the team adapts its approach in a way that accomplishes an increment of work. As each day passes, continual self- organization and collaboration move the team toward a completed software increment.

4.7.1.5 Coordination and Communication Issues

There are many reasons that software projects get into trouble. The scale of many development efforts is large, leading to complexity, confusion, and significant difficulties in coordinating team members. Uncertainty is common, resulting in a continuing stream of changes that ratchets the project team. Interoperability has become a key characteristic of many systems. New software must communicate with existing software and conform to predefined constraints imposed by the system or product.

These characteristics of modern software-scale, uncertainty, and interoperability— are facts of life. To deal with them effectively, you must establish effective methods for coordinating the people who do the work. To accomplish this, mechanisms for formal and informal communication among team members and between multiple teams must be established. Formal communication is accomplished through "writ- ing, structured meetings, and other relatively non-interactive and impersonal com- munication channels" [Kra95]. Informal communication is more personal. Members of a software team share ideas on an ad hoc basis, ask for help as problems arise, and interact with one another on a daily basis.

4.7.2Product

As the name inferred, this is the deliverable or the result of the project. The project manager should clearly define the product scope to ensure a successful result, control the team members, as well technical hurdles that he or she may encounter during the building of a product. The product can consist of both tangible or intangible such as shifting the company to a new place or getting a new software in a company.

A software project manager is confronted with a dilemma at the very beginning of a software project. Quantitative estimates and an organized plan are required, but solid information is unavailable. A detailed analysis of software requirements would provide necessary information for estimates, but analysis often takes weeks or even months to complete. We must examine the product and the problem it is intended to solve at the very beginning of the project. At a minimum, the scope of the product must be established and bounded.

4.7.2.1 Software Scope

The first software project management activity is the determination of software scope. Scope is defined by answering the following questions:

Context: How does the software to be built fit into a larger system, product, or business context, and what constraints are imposed as a result of the context?.

Information objectives: What customer-visible data objects are produced as output from the software? What data objects are required for input?

Function and performance: What function does the software perform to transform input data into output? Are any special performance characteristics to be addressed?

Software project scope must be unambiguous and understandable at the manage- ment and technical levels. A statement of software scope must be bounded. That is, quantitative data (e.g., number of simultaneous users, target environment, maxi- mum allowable response time) are stated explicitly, constraints and/or limitations (e.g., product cost restricts memory size) are noted, and mitigating factors (e.g., desired algorithms are well understood and available in Java) are described.

4.7.2.2 Problem Decomposition

Problem decomposition, sometimes called partitioning or problem elaboration, is an activity that sits at the core of software requirements analysis. During the scoping activity no attempt is made to fully decompose the problem.

Rather, decomposition is applied in two major areas:

(1) the functionality and content (information) that must be delivered and

(2) the process that will be used to deliver it.

Human beings tend to apply a divide-and-conquer strategy when they are con- fronted with a complex problem. Stated simply, a complex problem is partitioned into smaller problems that are more manageable. This is the strategy that applies as project planning begins. Software functions, described in the statement of scope, are evaluated and refined to provide more detail prior to the beginning of estimation. Because both cost and schedule estimates are functionally oriented, some degree of decomposition is often useful. Similarly, major content or data objects are decomposed into their constituent parts, providing a reasonable under- standing of the information to be produced by the software.

As an example, consider a project that will build a new word-processing product. Among the unique features of the product are continuous voice as well as virtual keyboard input via a multitouch screen, extremely sophisticated "automatic copy edit" features, page layout capability, automatic indexing and table of contents, and others. The project manager must first establish a statement of scope that bounds these features (as well as other more mundane functions such as editing, file management, and document production). For example, will continuous voice input re- quire that the product be "trained" by the user? Specifically, what capabilities will the copy edit feature provide? Just how sophisticated will the page layout capability be and will it encompass the capabilities implied by a multitouch screen?

As the statement of scope evolves, a first level of partitioning naturally occurs. The project team learns that the marketing department has talked with potential customers and found that the following functions should be part of automatic copy editing:

(1) spell checking,

(2) sentence grammar checking,

(3) reference checking for large documents (e.g., Is a reference to a bibliography entry found in the list of entries in the bibliography?),

(4) the implementation of a style sheet feature that imposed consistency across a document, and

(5) section and chapter reference validation for large documents. Each of these features represents a sub function to be implemented in software. Each can be further refined if the decomposition will make planning easier.

4.7.3Process

In every planning, a clearly defined process is the key to the success of any product. It regulates how the team will go about its development in the respective time period. The Process has several steps involved like, documentation phase, implementation phase, deployment phase, and interaction phase.

The team must decide which process model is most appropriate for

(1) the customers who have requested the product and the people who will do the work,

(2) the characteristics of the product itself, and

(3) the project environment in which the software team works. When a process model has been selected, the team then defines a preliminary project plan based on the set of process framework activities. Once the preliminary plan is established, process decomposition begins. That is, a complete plan, reflecting the work tasks required to populate the framework activities must be created. We explore these activities briefly in the sections that follow and present a more detailed view in Chapter 26.

4.7.3.1 Melding the Product and the Process

Project planning begins with the melding of the product and the process. Each function to be engineered by your team must pass through the set of framework activities that have been defined for your software organization.

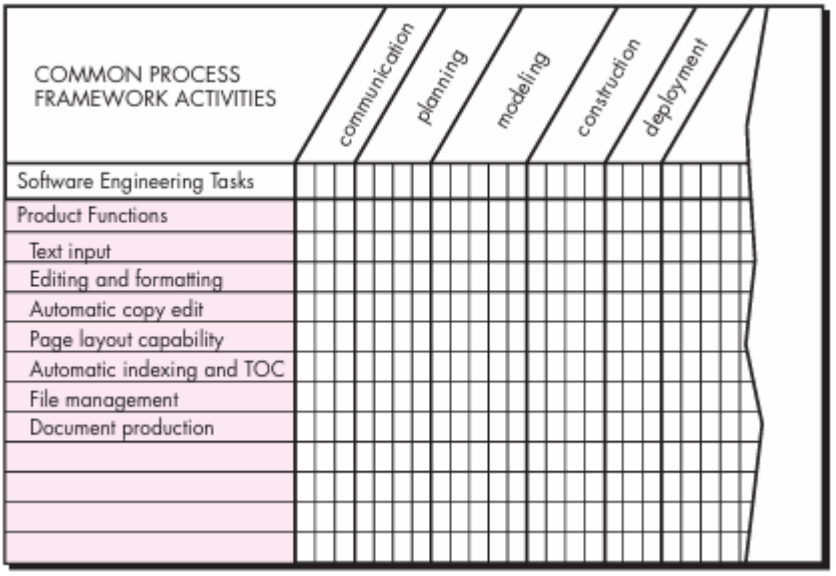

Fig 4.1 Melding the problem and process

4.7.3.1ProcessDecomposition

A software team should have a significant degree of flexibility in choosing the soft- ware process model that is best for the project and the software engineering tasks or that populate the process model once it is chosen. A relatively small project that is on similar to past efforts might be best accomplished using the linear sequential approach. If the deadline is so tight that full functionality cannot reasonably be delivered, an incremental strategy might be best.

Similarly, projects with other characteristics (e.g., uncertain requirements, breakthrough technology, difficult customers, significant reuse potential) will lead to the selection of other process models."

Once the process model has been chosen, the process framework is adapted to it. In every case, the generic process framework discussed earlier can be used. It will work for linear models, for iterative and incremental models, for evolutionary models, and even for concurrent or component assembly models. The process framework is invariant and serves as the basis for all work performed by a soft- ware organization.

But actual work tasks do vary. Process decomposition commences when the project manager asks, "How do we accomplish this framework activity?" For example, a small, relatively simple project might require the following work tasks for the communication activity:

1.Develop list of clarification issues.
2. Meet with stakeholders to address clarification issues.
3. Jointly develop a statement of scope.
4. Review the statement of scope with all concerned.
5. Modify the statement of scope as required.

Now, consider a more complex project, which has a broader scope and more significant business impact. Such a project might require the following work tasks for the communication:

1. Review the customer request.
2. Plan and schedule a formal, facilitated meeting with all stakeholders.
3. Conduct research to specify the proposed solution and existing approaches.
4. Prepare a "working document" and an agenda for the formal meeting.
5. Conduct the meeting.
6. Jointly develop mini-specs that reflect data, functional, and behavioural features of the software. Alternatively, develop use cases that describe the software from the user's point of view.
7. Review each mini-spec or use case for correctness, consistency, and lack of ambiguity.
8. Assemble the mini-specs into a scoping document.
9. Review the scoping document or collection of use cases with all

10. concerned.
11. Modify the scoping document or use cases as required.

Both projects perform the framework activity that we call communication, but the first project team performs half as many software engineering work tasks as the second.

4.7.3Project

The last and final P in software project planning is Project. It can also be considered as a blueprint of process. In this phase, the project manager plays a critical role. They are responsible to guide the team members to achieve the project's target and objectives, helping & assisting them with issues, checking on cost and budget, and making sure that the project stays on track with the given deadlines.

In order to manage a successful software project, engineer have to understand what can go wrong so that problems can be avoided. In an excellent paper on software projects, John Reel defines 10 signs that indicate that an information systems project is in jeopardy:

1. Software people don't understand their customer's needs.
2. The product scope is poorly defined.
3. Changes are managed poorly.
4. The chosen technology changes.
5. Business needs change [or are ill defined].
6. Deadlines are unrealistic.
7. Users are resistant.
8. Sponsorship is lost [or was never properly obtained].
9. The project team lacks people with appropriate skills.
10. Managers [and practitioners] avoid best practices and lessons learned.

Jaded industry professionals often refer to the 90-90 rule when discussing particularly difficult software projects: The first 90 percent of a system absorbs 90 percent of the allotted effort and time. The last 10 percent takes another 90 percent of the allotted effort and time. The seeds that lead to the 90-90 rule are contained in the signs noted in the preceding list.

Reel suggests a five-part commonsense approach to software projects:

1. Start on the right foot: This is accomplished by working hard (very hard) to understand the problem that is to be solved and then setting realistic objectives and expectations for everyone who will be involved in the project. It is reinforced by building the right team and giving the team the autonomy, authority, and technology needed to do the job.

2. Maintain momentum: Many projects get off to a good start and then slowly disintegrate. To maintain momentum, the project manager must provide incentives to keep turnover of personnel to an absolute minimum, the team should emphasize quality in every task it performs, and senior management should do everything possible to stay out of the team's way.?.

3. Track progress. For a software project, progress is tracked as work products (e.g., models, source code, sets of test cases) are produced and approved (using technical reviews) as part of a quality assurance activity. In addition, software process and project measures can be collected and used to assess progress against averages developed for the software development organization.

4. Make smart decisions. In essence, the decisions of the project manager and the software team should be to "keep it simple." Whenever possible, decide to use commercial off-the-shelf software or existing software components or patterns, decide to avoid custom interfaces when standard approaches are available, decide to identify and then avoid obvious risks, and decide to allocate more time than you think is needed to complex or risky tasks (you'll need every minute).

5. Conduct a postmortem analysis. Establish a consistent mechanism for extracting lessons learned for each project. Evaluate the planned and actual schedules, collect and analyze software project metrics, get feedback from team members and customers, and record findings in written form.

4.8 W^{5}HH PRINCIPLE

In an excellent paper on software process and projects, Barry Boehm states: "you need an organizing principle that scales down to provide simple plans for simple projects." Boehm suggests an approach that addresses project objectives, milestones and schedules, responsibilities, management and technical approaches, and required resources. He calls it the WSHH Principle, after a series of questions that lead to a definition of key project characteristics and the resultant project plan:

Why is the system being developed?

All stakeholders should assess the validity of business reasons for the software work. Does the business purpose justify the expenditure of people, time, and money?

What will be done?

The task set required for the project is defined.

When will it be done?

The team establishes a project schedule by identifying when project tasks are to be conducted and when milestones are to be reached.

Who is responsible for a function?

The role and responsibility of each member of the software team is defined.

Where are they located organizationally?

Not all roles and responsibilities reside within software practitioners. The customer, users, and other stakeholders also have responsibilities.
How will the job be done technically and managerially?

Once product scope is established, a management and technical strategy for the project must be defined.

How much of each resource is needed?

The answer to this question is derived by developing estimatesbased on answers to earlier questions.

Boehm's WSHH Principle is applicable regardless of the size or complexity of a software project.

4.9 CRITICAL PRACTICES

The Airlie Council has developed a list of "critical software practices for performance-based management." These practices are "consistently used by, and considered critical by, highly successful software projects and organizations whose 'bottom line' performance is consistently much better than industry averages".

Critical practices include: metric based project management, empirical cost and schedule estimation, earned value tracking, defect tracking against quality targets, and people aware management.