

UNIT IV

Project Organizations

Project Organizations and Responsibilities

Software Lines of Business and Project Teams

Software lines of business and project teams have different motivations. **Software lines of business** focus on return on investment, new business opportunities, market diversification, and profitability.

Project teams mainly focus on the cost, schedule, and quality of specific project deliverables.

Software professionals in both types of organizations are motivated by **career growth, job satisfaction, and the opportunity to make a difference.**

Organizational Structure

In the past, most attention was given to the **project**, because software is developed and delivered at the project level.

However, projects usually focus on their own cost, schedule, and quality. They may not invest in technologies or services that do not directly benefit their project.

Therefore, organizations must provide the necessary **infrastructure and common processes** to support software projects.

Organizational Roles and Responsibilities

The organizational structure should define the **roles, relationships, and responsibilities** needed for both the software line of business and individual projects.

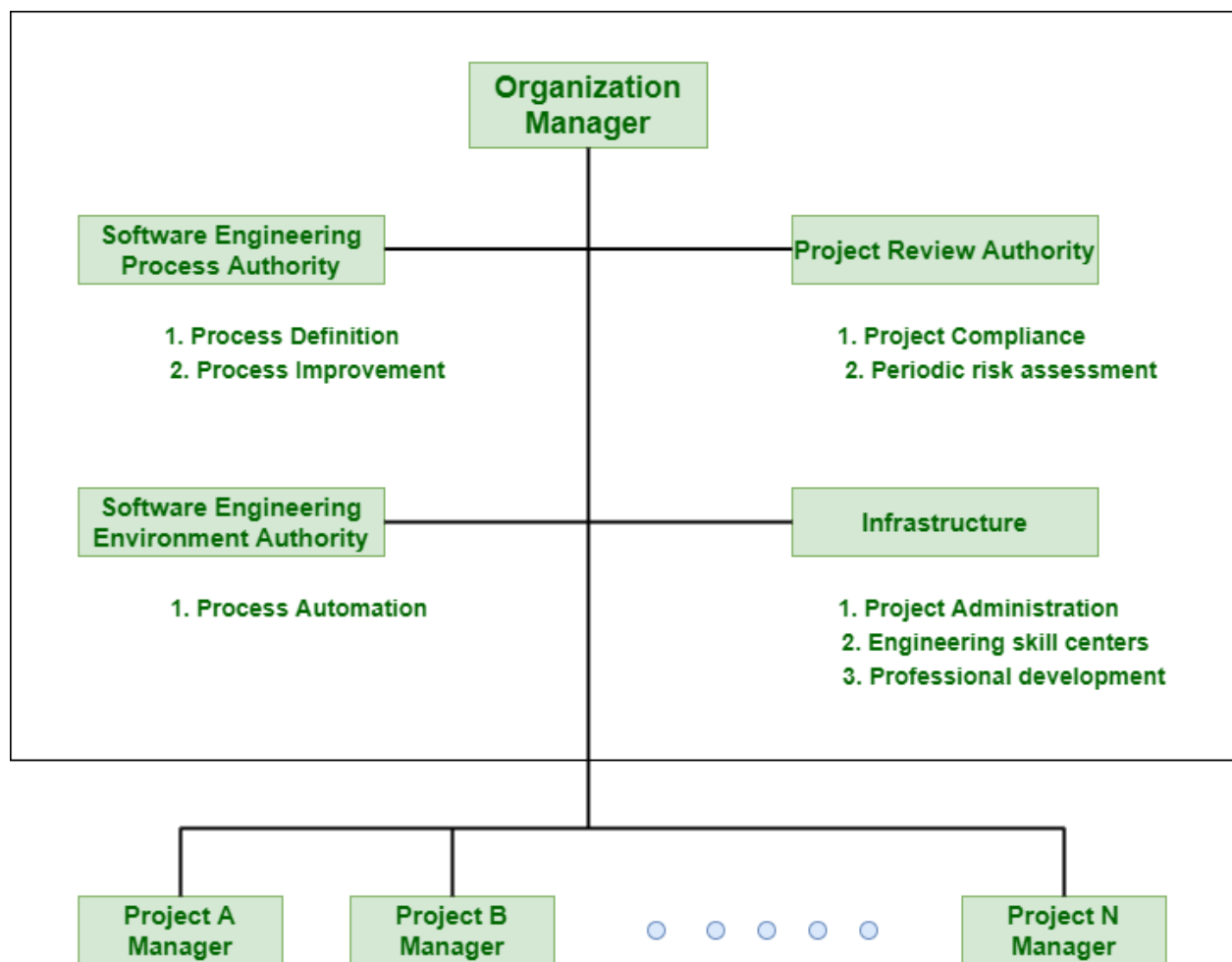
A fixed organizational hierarchy is not suitable for every organization. The structure should depend on the **domain, scale, culture, and people** involved.

Tailoring the Organization

The suggested organizational structure should be treated as a **default starting point**. According to the situation, roles may be **split or merged**, and projects or teams may be organized differently.

These organizational guidelines include many recurring practices found in successful software projects and provide a useful framework for most software organizations.

Line-Of-Business Organizations



Default roles in a Software Line-of-Business Organization

Line-of-Business Organizations

A line-of-business organization is an organizational structure designed for a particular business area. This structure can be modified according to the needs of the organization.

The main features of a default line-of-business organization are process definition and maintenance, process automation, and organizational roles. Process definition is specific to a particular line of business where common processes are useful. Process automation is also an important organizational responsibility.

The organizational roles may be performed by a single person or several teams, depending on the size of the organization. Small organizations may need only one person, while large organizations may require many people.

Software Engineering Process Authority

The Software Engineering Process Authority (SEPA) facilitates the exchange of process information and guidance between the organization and project practitioners.

SEPA is responsible for maintaining the organization's current process maturity assessment and plans for future process improvement. It also helps initiate and periodically assess project processes.

SEPA has responsibility and accountability for process definition and maintenance, including modification, improvement, and technology insertion. It may be an individual, a general manager, or a team of representatives.

Project Review Authority

The Project Review Authority (PRA) is responsible for ensuring that a software project follows the organization's business and software policies, practices, and standards.

The PRA reviews the project's compliance with contractual and organizational obligations. It also reviews progress, quality, cost, schedule, risk, deliverables, and financial performance.

Software Engineering Environment Authority

The Software Engineering Environment Authority (SEEA) is responsible for process automation in the organization.

SEEA maintains the organization's standard software development environment, trains projects to use it, and maintains organization-wide reusable assets.

A common environment allows tools, techniques, and training to be used across multiple projects. The environment may be customized when necessary, while maintaining a common default solution for most projects.

Infrastructure

An organization's infrastructure provides human-resource support, project-independent research and development, and other capital software engineering assets.

The main components of organizational infrastructure are project administration, engineering skill centers, and professional development.

Project administration includes time accounting, contracts, pricing, terms and conditions, and integration with corporate information systems.

Engineering skill centers provide custom tool repositories and maintenance, bid and proposal support, and independent research and development.

Professional development includes internal training, personnel recruitment, skills database maintenance, literature and asset libraries, and technical publications.

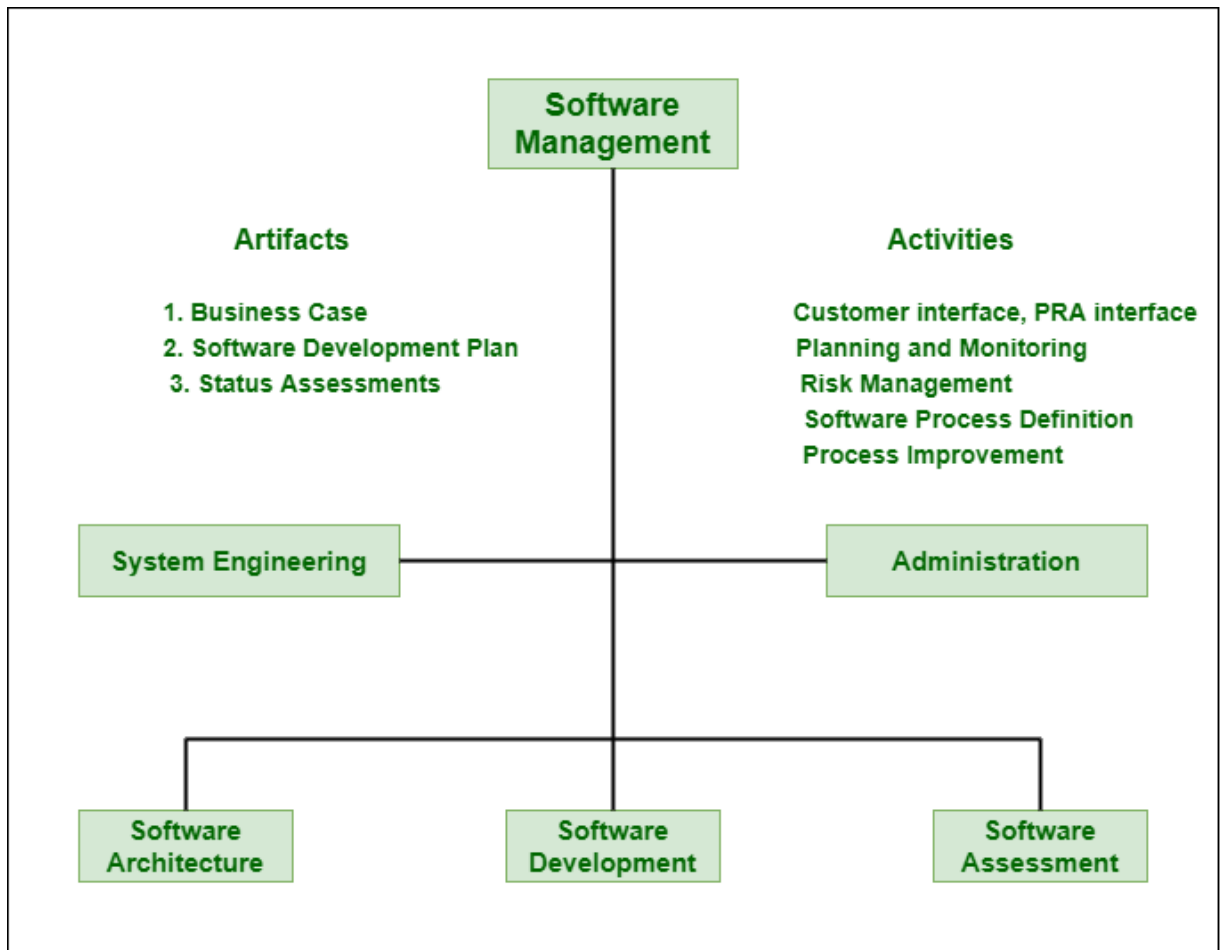
An organizational service center promotes a standard environment that is funded by the line of business and maintained as a capital asset for projects.

SEPA is responsible for process definition and improvement, whereas SEEA is responsible for process automation.

In mature software organizations, software development environments, processes, and tools should be treated as organizational assets, similar to capital equipment. They should therefore be funded using capital resources.

Funding may be provided through overhead or general administrative costs, or through project billing based on usage. Organizations using this capital-investment approach are generally mature, large-scale software developers with stable processes and long-term relationships with software tool vendors.

PROJECT ORGANIZATIONS



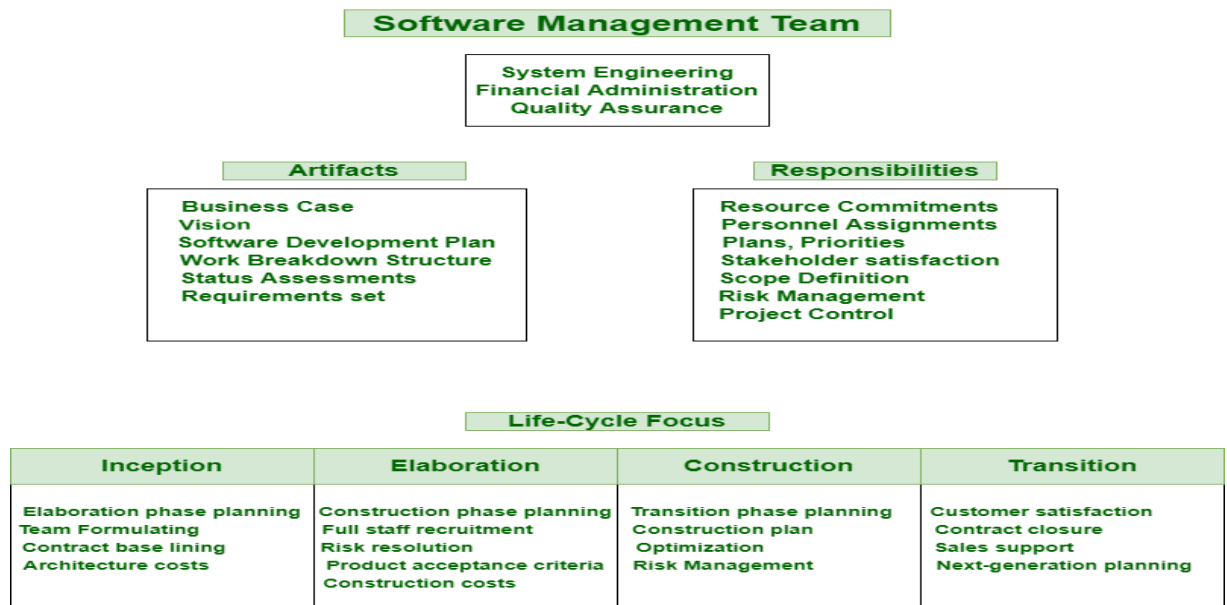
Default Project Organization and Responsibilities

The main features of the default organization are as follows:

- The project management team is an active participant, responsible for producing as well as managing.
- The architecture team is responsible for real artifacts and for the integration of Components, not just for staff functions.

- The development team owns the component construction and maintenance activities.
- The assessment team is separate from development.
- Quality is everyone's into all activities and checkpoints.
- Each team takes responsibility for a different quality perspective.

Software Management Team:



Most projects are over constrained because schedule, cost, functionality, and quality expectations are closely related. They require continuous negotiation among stakeholders with different goals.

The software management team carries the responsibility of delivering suitable conditions to all stakeholders. The software project management team therefore spends its effort maintaining a proper balance throughout the project life cycle. The team is responsible for planning the effort, conducting the plan, and adapting the plan when requirements or the design change.

It takes ownership of resource management and project scope and sets operational priorities throughout the project life cycle.

At an abstract level, these activities involve managing the expectations of all stakeholders throughout the project life cycle.

The software management team also takes ownership of all aspects of quality. It maintains a balance among these aspects so that the overall solution is adequate for all stakeholders and optimal for as many stakeholders as possible.

Software Management Activities

The major artifacts are the business case, vision, software development plan, work breakdown structure, status assessments, and requirements set.

The main software management activities include systems engineering, financial administration, and quality assurance.

The major responsibilities include resource commitments, personnel assignments, plans and priorities, stakeholder satisfaction, scope definition, risk management, and project control.

Life-Cycle Focus

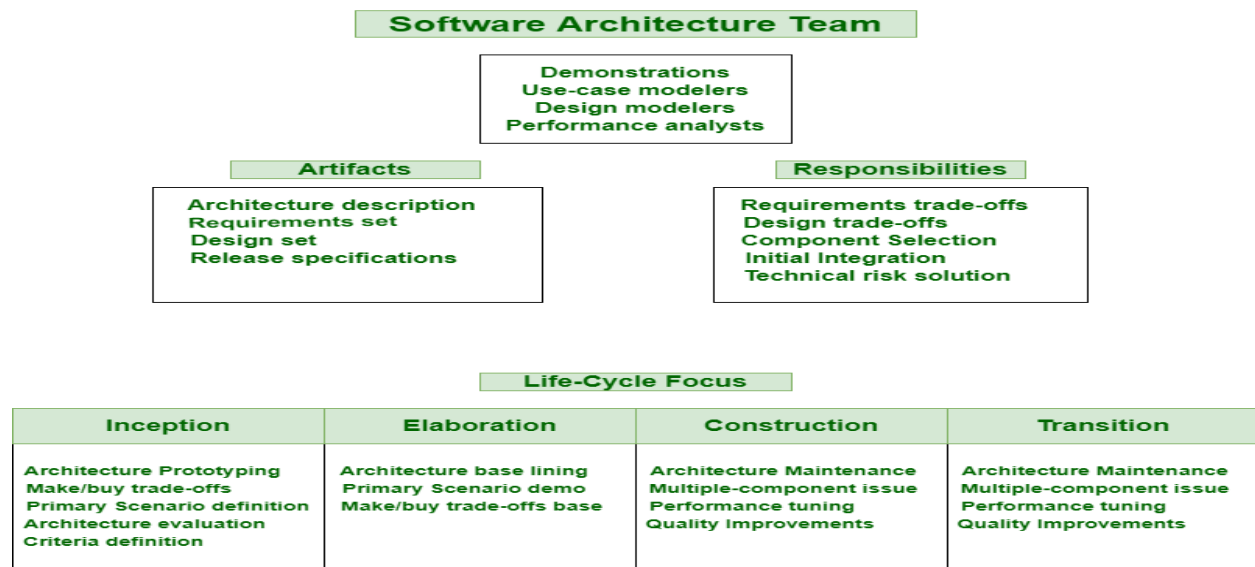
During Inception, the focus is on elaboration phase planning, team formulation, contract baselining, and architecture costs.

During Elaboration, the focus is on construction phase planning, full staff recruitment, risk resolution, product acceptance criteria, and construction costs.

During Construction, the focus is on transition phase planning, construction plan optimization, and risk management.

During Transition, the focus is on customer satisfaction, contract closure, sales support, and next-generation planning.

Software Architecture Team:



The software architecture team is responsible for defining and maintaining the software architecture. It specifies the complete bill of materials for the software and performs the engineering needed for important make/buy trade-offs. This helps make the construction or assembly cost of custom components predictable.

The architecture team is very important to a project. It provides a framework for team communication, helps achieve system-wide quality, and supports application implementation. A good architecture team can help an average development team succeed, while a weak architecture may cause even an expert development team to fail.

Role During the Life Cycle

During the inception and elaboration phases, the project is mainly handled by the software management team and the software architecture team. The software development and assessment teams mainly provide support during these phases.

When the construction phase begins, the architecture enters maintenance mode. Only a minimum level of effort is required to maintain continuity of the engineering knowledge and architecture.

Architecture Artifacts

The main artifacts of the software architecture team are architecture description, requirements set, design set, and release specifications.

The team also works with demonstrations, use case models, design models, and performance analysis to develop and evaluate the architecture.

Responsibilities

The main responsibilities include requirements trade-offs, design trade-offs, component selection, initial integration, and technical risk resolution.

Life-Cycle Focus

During inception, the team performs architecture prototyping, make/buy trade-offs, primary scenario definition, and defines architecture evaluation criteria.

During elaboration, the focus is on architecture baselining, primary scenario demonstration, and make/buy trade-off baselining.

During construction, the focus is on architecture maintenance, multiple-component issue resolution, performance tuning, and quality improvements.

During transition, the team continues architecture maintenance, resolves multiple-component issues, performs performance tuning, and improves quality.

Required Expertise

The architecture team requires domain experience to produce an acceptable design view and use case view.

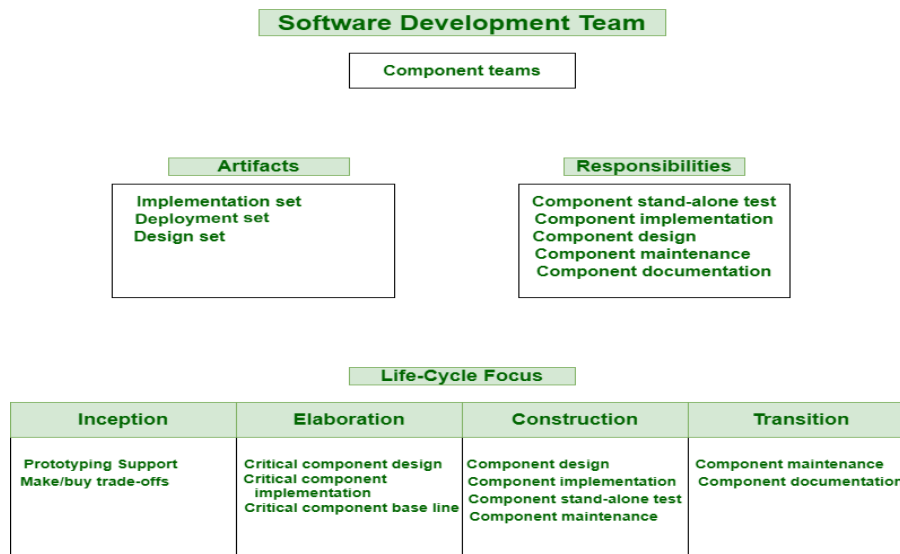
It also requires software technology experience to produce the process view, component view, and deployment view.

System-Level Quality

The architecture team is responsible for system-level quality attributes such as reliability, performance, and maintainability.

These qualities involve multiple components and depend on how well the components are integrated. Therefore, the architecture team decides how major multiple-component design issues are resolved

Software Development Team:



The software development team is the most application-specific group in a software project. It consists of several component teams, where each team works on components requiring similar skills.

Artifacts

The main artifacts produced by the software development team are the design set, implementation set, and deployment set.

Responsibilities

The team is responsible for component design, implementation, stand-alone testing, maintenance, and documentation.

Life-Cycle Focus

Inception: The team provides prototyping support and helps in making trade-offs.

Elaboration: The focus is on critical component design, critical component implementation and testing, and establishing the critical component baseline.

Construction: The focus is on component design, implementation, stand-alone testing, and maintenance.

Transition: The focus is on component maintenance and documentation.

Skill Sets

Commercial Component: Specialists with knowledge of commercial components used in the system architecture.

Database: Specialists experienced in the organization, storage, and retrieval of data.

Graphical User Interfaces: Specialists experienced in display organization, data presentation, and user interaction.

Operating Systems and Networking: Specialists experienced in executing software objects on networked hardware resources, including initialization, synchronization, resource sharing, name-space management, reconfiguration, termination, and interobject communication.

Domain Applications: Specialists experienced in algorithms, application processing, and business rules specific to the system.

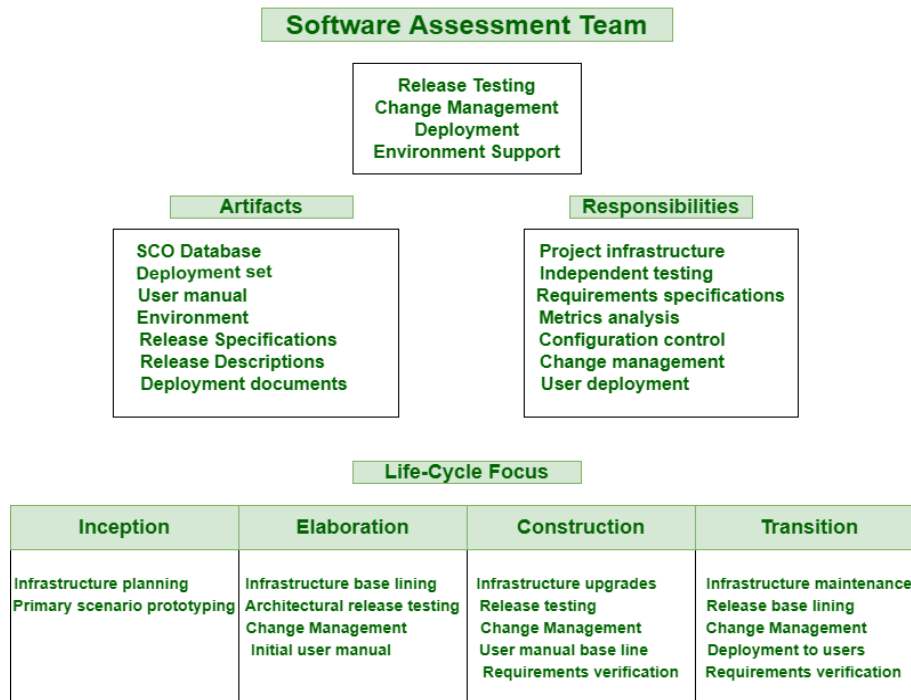
Quality Responsibility

The software development team is responsible for the quality of individual components, including their development, testing, and maintenance.

Component tests should be self-documented and repeatable. They are maintained like other source code and can be used for automated regression testing.

The development team also decides how to resolve design or implementation issues related to an individual component.

Software Assessment Team:



A software assessment team is responsible for assessing software quality throughout the project life cycle.

An independent team is used for two main reasons. First, it provides an independent quality perspective. It prevents the developers' ownership of the software from affecting the quality assessment.

The second reason is concurrency of activities. Software development and testing activities can be performed in parallel. Change management, test planning, and test scenario development can proceed along with design and development.

Software Assessment Activities

The software assessment team works with important artifacts such as the deployment set, SCO database, user manual, environment, release specifications, release descriptions, and deployment documents.

Its major responsibilities include project infrastructure, independent testing, requirements verification, metrics analysis, configuration control, change management, and user deployment.

The major assessment activities are release testing, change management, deployment, and environment support.

Life-Cycle Focus

During Inception, the focus is on infrastructure planning and primary scenario prototyping.

During Elaboration, the focus is on infrastructure baseline, architecture release testing, change management, and the initial user manual.

During Construction, the focus is on infrastructure upgrades, release testing, change management, user manual baseline, and requirements verification.

During Transition, the focus is on infrastructure maintenance, release baselining, change management, deployment to users, and requirements verification.

Release-Based Testing

A modern process uses use-case-oriented or capability-based testing. Testing may involve many components and is organized as a sequence of builds.

Two important artifacts are used:

Release specification defines the plan and evaluation criteria for a release.

Release description records the results of a release.

Each release may contain several components, including incomplete components, because integration continues throughout development.

The evaluation criteria describe what the customer can expect at a major milestone. The release description provides the results of testing.

The final iterations generally become similar to acceptance testing and contain details similar to conventional test plans, procedures, and reports.

Development of Testing Artifacts

Early releases contain brief and abstract evaluation criteria. Later releases become more detailed and rigorous, including completeness and traceability information.

For use-case testing, test components should be developed similar to component test cases. Instead of separate test procedure documents, self-documenting test scenarios can be created as software programs.

These test scenarios are subject to change management and are maintained up-to-date for automated regression testing.

Component Testing

Some component tests may become part of the evaluation criteria, and their results are documented in release descriptions.

Some components may receive only informal testing by the development team. Other components require formal testing through higher-level evaluation criteria.

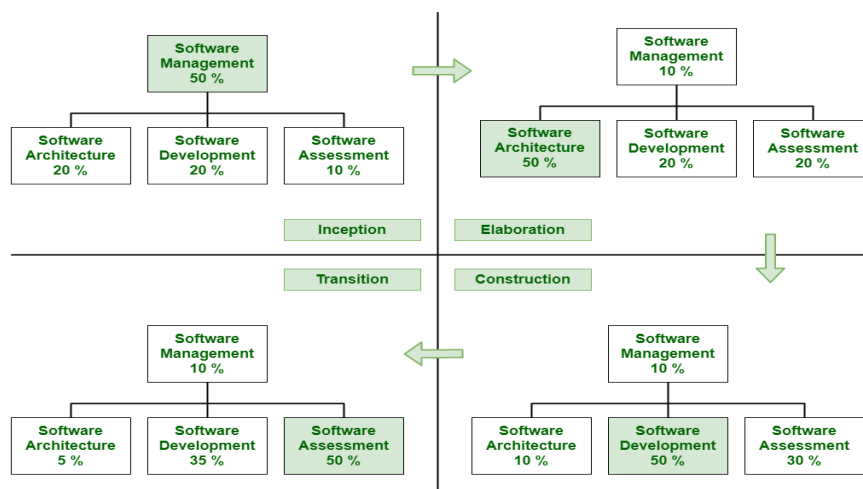
The assessment team decides whether a component should be tested formally to verify requirements or tested as part of capability testing.

Responsibility of Assessment Team

The assessment team is responsible for the quality of baseline releases with respect to requirements and customer expectations.

Therefore, it must identify and expose quality issues that affect customer expectations, even when those expectations are not explicitly captured in the requirements

EVOLUTION OF ORGANIZATIONS



Evolution of Organizations

The project organization represents the architecture of the project team. It evolves according to the project plan defined in the Work Breakdown Structure (WBS).

The team focus changes during each phase of the software life cycle. About 50% of the staff may be assigned to one major set of activities in each phase.

Inception Team

The Inception team is mainly focused on planning. Other teams provide sufficient support so that the plans represent the consensus of all perspectives.

Elaboration Team

The Elaboration team is an architecture-focused organization. The major project activities are handled by the Software Architecture team. Software Development and Software Assessment support the architecture team to achieve a stable architecture baseline.

Construction Team

The Construction team is a fairly balanced organization. Most activities are performed by the Software Development and Software Assessment teams.

Transition Team

The Transition team is a customer-focused organization. Customer feedback drives the deployment activities during this phase.

Team Structure Planning

It is important to define the details of subteams, responsibilities, and work packages. However, these details should be finalized only after the WBS planning details become stable.

Defining lower-level team structures too early may result in serious downstream inefficiencies.

PROCESS AUTOMATION

Process Automation

Many software development organizations evolve mature processes to improve the **predictability of software management** and the performance of their software business. This includes improving product quality, time to market, return on investment, and productivity.

Process definition and tailoring are necessary, but **process automation** is also required to modernize software development and complete projects profitably.

The level of automation depends on the **scale of the development effort**. The techniques, training, time scales, acceptance criteria, and level of automation vary according to the size and complexity of the project.

Development Environment and Infrastructure

Software organizations need to integrate their **environment and infrastructure** for software development. This may involve different tools, information repositories, vendors, platforms, programming languages, and process assumptions.

Integrating these different components can become more difficult than expected.

Automating the development process and establishing an infrastructure are important activities during the **engineering stage of the life cycle**. These activities include **tool selection, custom tool smithing, and process automation**.

The development environment should also evolve into a **maintenance environment** so that it can support the software throughout its life cycle.

The environment that provides process automation is a **tangible artifact** and is important to the life-cycle cost of the developed system. Therefore, the environment must be treated as a **first-class artifact of the process**.

Stakeholders should have access to the environment resources so that they can interact with the development team and add value to the development process.

Round-trip engineering and **integrated environments** provide greater freedom to evolve technical artifacts. **Metrics automation** is also important for effective project control.

Process automation and change management are especially important in an **iterative process**, because changes occur continuously during iterative development.

Three Levels of Process

There are three levels of process that require different levels of automation:

1. Metaprocess

The **metaprocess** refers to an organization's policies, procedures, and practices for managing a software-intensive line of business.

The automation support for the metaprocess is called **infrastructure**.

An infrastructure contains preferred tools, artifact templates, microprocess and macroprocess guidelines, a project performance repository, organizational skill-set information, and a library of previous project plans and results.

2. Macroprocess

The **macroprocess** refers to a project's policies, procedures, and practices for producing a complete software product within specified **cost, schedule, and quality constraints**.

The automation support for the project's process is called an **environment**.

An environment is a specific collection of tools used to produce a specific set of artifacts according to a specific project plan.

3. Microprocess

The **microprocess** refers to a project team's policies, procedures, and practices for achieving an artifact of the software process.

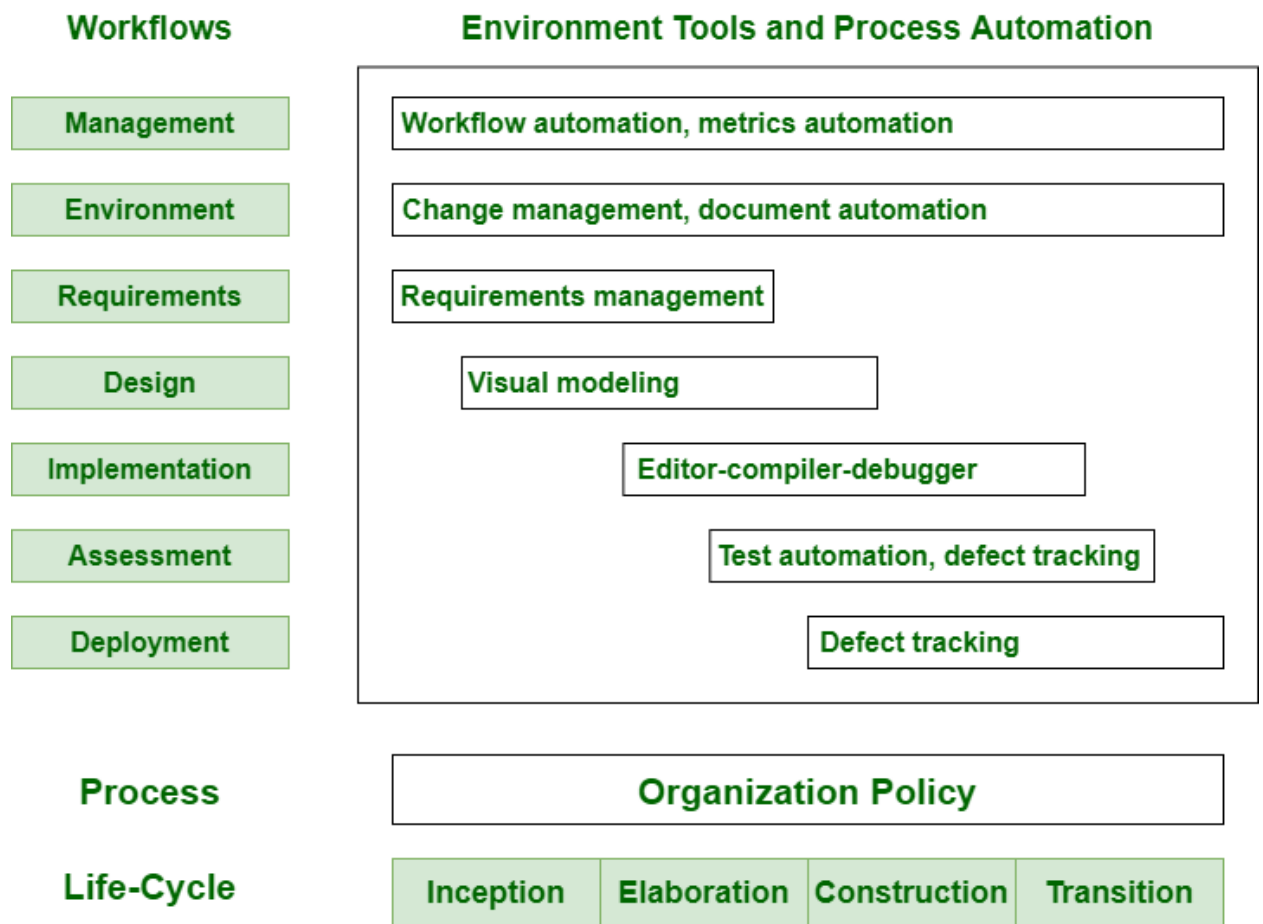
The automation support for generating an artifact is generally called a **tool**.

Typical tools support requirements management, visual modeling, compilation, editing, debugging, change management, metrics automation, document automation, test automation, cost estimation, and workflow automation.

Thus, **infrastructure supports the metaprocess, environment supports the macroprocess, and tools support the microprocess.**

The focus of process automation is the **workflow of the project environment**. However, the infrastructure of the parent organization and the tool-building blocks are also important prerequisites.

TOOLS: AUTOMATION BUILDING BLOCKS



Automation and Tool components that support the process workflows

TOOLS: AUTOMATION BUILDING BLOCKS

Many tools are available to automate the software development process. These tools provide the environment needed to support different process workflows.

Each workflow has a specific need for automation. Some tools are used to generate artifacts, while others are used for tracking and management.

Management

The management workflow can be supported by workflow automation and metrics automation.

Software cost estimation tools and WBS tools help in generating project planning artifacts. Workflow management tools and software project control panels help maintain the project status against the plan.

Automation also improves the collection and reporting of project metrics.

Environment

Configuration management and version control are essential in modern iterative development.

The environment should provide change management automation to manage changes in software artifact baselines.

Requirements

In the traditional approach, system requirements are divided into subsystem, component, and unit requirements. This approach requires considerable effort for detailed traceability.

In a modern process, system requirements are captured in the vision statement. Lower-level requirements are organized according to iterations and represented through evaluation criteria, use cases, and other textual objectives.

The vision statement represents the contract between the development group and the buyer. It evolves throughout the life cycle and should be understandable to the buyer.

Evaluation criteria are captured in release specification artifacts. They represent the objectives of a particular iteration.

Requirements can evolve along with the architecture and application increments. This helps the customer and developer understand priorities and the cost, schedule, and performance trade-offs.

Requirements Management Support

The requirements approach uses both textual and model-based representations. Therefore, the environment should provide document automation and visual modeling.

The environment must also manage and track changes to requirements and present them in a human-readable form.

Traceability between requirements and other artifacts should be automated. Requirements should have well-defined traceability to test artifacts so that compliance with requirements can be demonstrated.

However, strong traceability between requirements and all technical artifacts can be difficult, especially in component-based architectures.

Design

Requirements, design, implementation, and assessment tools are generally used together.

The main support required for the design workflow is visual modeling. It is used to capture design models, present them in human-readable form, and transform them into source code.

An architecture-first and demonstration-based process is supported by existing architecture components and middleware.

Implementation

The implementation workflow mainly depends on a programming environment, including an editor, compiler, debugger, linker, and run-time environment.

It should also be integrated with change management, visual modeling, and test automation tools. This supports productive and iterative development.

Assessment and Deployment

The assessment workflow requires test automation and test management capabilities.

Testing and document production should be mostly automated to increase change freedom.

Defect tracking is an important assessment tool. It supports automation of metrics and control of release baselines.

Defect tracking also supports the deployment workflow throughout the life cycle.

THE PROJECT ENVIRONMENT

The Project Environment

The project environment evolves through three states: prototyping environment, development environment, and maintenance environment.

1. Prototyping Environment

The prototyping environment provides an architecture testbed for evaluating project architectures during the inception and elaboration phases.

It supports performance trade-offs and technical risk analysis, make/buy trade-offs and feasibility studies, and fault tolerance and dynamic reconfiguration trade-offs.

It also helps analyse the risks of moving to full-scale implementation and supports the development of test scenarios, tools, and instrumentation for analysing requirements.

2. Development Environment

The development environment contains a complete set of development tools required to support different process workflows.

It should also support round-trip engineering to the maximum possible extent.

3. Maintenance Environment

The maintenance environment normally corresponds to a mature version of the development environment.

In some cases, it may be a subset of the development environment delivered as one of the project's end products.

Important Environment Disciplines

The transition to a mature software process creates challenges in controlling concurrent activities, progress, and quality.

A highly integrated environment is therefore required for effective management control of the process.

Round-Trip Engineering

Tools must be integrated to maintain consistency and traceability. This requirement is known as round-trip engineering and is important for iterative development.

Change Management

Change management must be automated and enforced to manage multiple iterations and allow changes when required. Change is a fundamental principle of iterative development.

Organizational Infrastructure

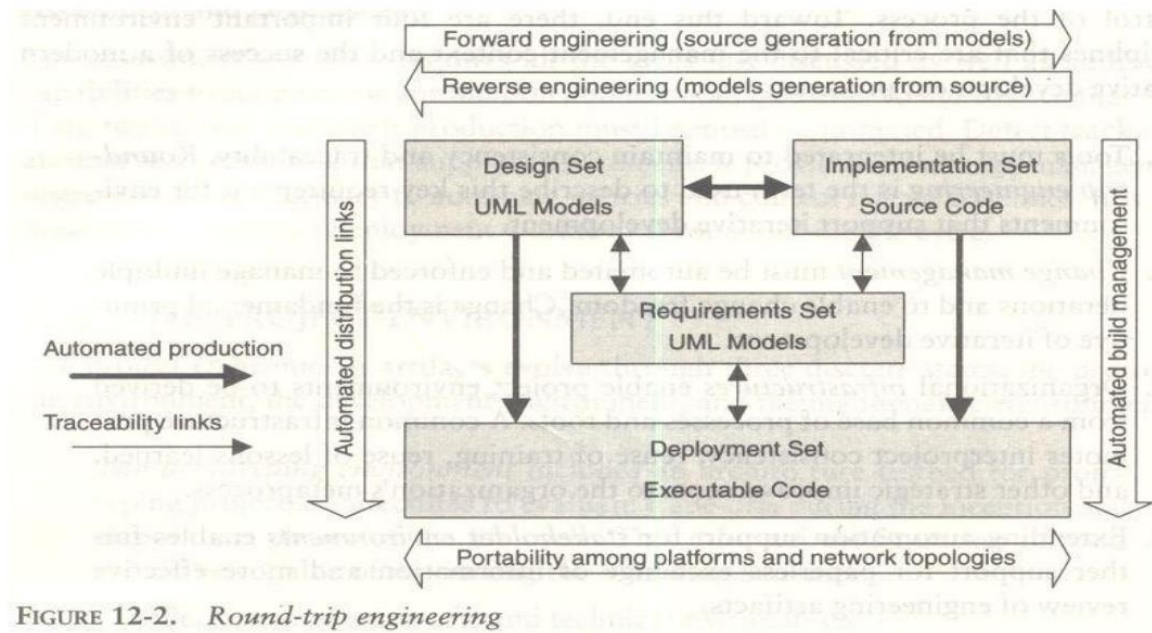
Organizational infrastructures allow project environments to use a common base of processes and tools.

This promotes interproject consistency, reuse of training and lessons learned, and strategic improvements to the organization's metaprocess.

Stakeholder Environment

Extended automation support for stakeholder environments enables paperless exchange of information and more effective review of engineering artifacts.

ROUND TRIP ENVIRONMENT:



Round-Trip Engineering

As software development uses different information sets for engineering artifacts, automation support is needed for efficient and error-free transfer of data between artifacts.

Round-trip engineering provides the environment support needed to maintain consistency among engineering artifacts.

Forward and Reverse Engineering

The automated translation of design models into source code is called forward engineering. The reverse process, in which models are generated from source code, is called reverse engineering.

The translation of design models into process or distribution models is also becoming easier through technologies such as ActiveX and CORBA (Common Object Request Broker Architecture).

Automation of Build Management

Compilers and linkers have long provided automation for converting source code into executable code.

Modern architecture use different components, platforms, and programming languages. Therefore, building and maintaining large collections of components requires configuration control and automated build management.

However, current environments do not provide complete automation. For example, automatic construction of test cases from use-case and scenario descriptions is still limited mainly to simple unit test scenarios.

Purpose of Round-Trip Engineering

The main purpose of round-trip engineering is to allow freedom in changing software engineering data sources.

Configuration control of technical artifacts is important for maintaining a consistent and error-free representation of the evolving software product.

Round-trip engineering does not require every transition to be bidirectional. Some information cannot be reconstructed in the reverse direction.

Round-Trip Engineering Model

The figure shows different engineering artifact sets. The Design Set contains UML models, while the Implementation Set contains source code.

The Requirements Set contains UML models related to requirements. The Deployment Set contains executable code.

The model supports automated production, traceability links, automated distribution links, and automated build management. It also supports portability among platforms and network topologies.

Completeness of Translation

Translation from one data source to another may not provide 100% completeness. For example, translating design models into C++ source code may provide only the structural and declarative parts of the source code. The generated code may still need additional details for particular object attributes or methods.

CHANGE MANAGEMENT:

Title: _____		
Description	Name: _____	Date: _____
	Project: _____	
Metrics	Category: _____ (0/1 error, 2 enhancement, 3 new feature, 4 other)	
Initial Estimate	Actual Rework Expended	
	Breakage: _____	Analysis: _____ Test: _____
	Rework: _____	Implement: _____ Document: _____
Resolution	Analyst: _____	
	Software Component: _____	
Assessment	Method: _____ (inspection, analysis, demonstration, test)	
Disposition	Tester: _____ Platforms: _____ Date: _____	
	State: _____	Release: _____ Priority: _____
	Acceptance: _____	Date: _____
	Closure: _____	Date: _____

FIGURE 12-3. *The primitive components of a software change order*

Change Management

Change management is as important to an iterative process as planning. Tracking changes in technical artifacts helps in understanding the actual technical progress and quality trends of a software product or interim release.

In conventional software management, baseline configuration management was mainly performed late in the life cycle. In a modern software process, requirements, design, and implementation artifacts are captured early using rigorous notations and are evolved through several generations. Therefore, change management becomes a fundamental activity throughout almost all phases of the software life cycle.

Software Change Orders

A **Software Change Order (SCO)** is the basic unit of software work authorized to create, modify, or make obsolete components within a configuration baseline.

SCOs provide an important mechanism for partitioning, allocating, and scheduling software work against an established baseline. They are also useful for assessing project progress and software quality.

By automating data entry and maintaining change records online, much of the administrative work associated with metrics reporting and change management can also be automated.

Level of an SCO

Choosing the correct level for writing an SCO is important. A change may involve a program unit, component, file, subsystem, new feature, defect resolution, or performance improvement.

Generally, an SCO should be written for a **single component** so that it can be assigned easily to one individual. If resolving a change requires two people from different teams, two separate SCOs should normally be created.

Basic Fields of an SCO

The basic fields of a Software Change Order are **Title, Description, Metrics, Resolution, Assessment, and Disposition.**

Title

The title is initially suggested by the person who originates the SCO and is finalized when it is accepted by the **Configuration Control Board (CCB)**. If the change originated from an external person such as a user, the title should also contain a reference to the external software problem report.

Description

The description contains the originator's name, date of origination, CCB-assigned SCO identifier, and relevant version identifiers of related support software.

The problem should be described with sufficient detail. Supporting information such as code extracts, display snapshots, error messages, or other useful data may be included to clearly identify the problem or required change.

Metrics

Metrics collected for every SCO are important for **planning, scheduling, and assessing quality improvement**.

Change categories are classified as **Type 0 – critical bug, Type 1 – bug, Type 2 – enhancement, Type 3 – new feature, and Type 4 – other**.

When an SCO is accepted, initial estimates are made for **breakage** and **rework**. Breakage represents the volume or extent of change, while rework represents the complexity or effort involved in making the change.

After resolving the SCO, actual breakage and actual rework effort are recorded. Rework effort is further divided into **analysis, implement, test, and document** activities.

The **analysis** item represents the staff hours used to understand, isolate, and debug the required change. The **implement** item represents the effort required to design and implement the solution. The **test** item records the effort spent testing the resolution, while the **document** item records the effort required to update related artifacts such as user manuals or release descriptions.

Breakage may be measured using **SLOC, function points, files, components, or classes**. For SLOC, a source-file comparison program can measure the differences

and provide an estimate of breakage. The required precision depends on the size of the change.

Resolution

The resolution field records the person responsible for implementing the change, the components changed, actual metrics, and a description of the change.

The component reference should normally be detailed enough to correspond approximately to the level of work allocated to an individual. A component assigned to an entire team is generally not sufficiently detailed.

Assessment

The assessment field specifies how the completed change is evaluated. The assessment technique may be **inspection, analysis, demonstration, or test**.

Where applicable, it should refer to existing test cases and any new test cases executed. It should also identify different test configurations such as platforms, topologies, and compilers.

Disposition

The **Configuration Control Board (CCB)** assigns a state to each SCO. **Proposed** means the SCO has been written and is waiting for CCB review. **Accepted** means it has been approved for resolution. **Rejected** means it is closed with a reason, such as being a duplicate, obsolete, not a problem, or already resolved by another SCO.

Archived means the SCO is accepted but postponed to a later release. **In Progress** means it has been assigned and is actively being resolved by the development organization. **In Assessment** means development has completed the resolution and the test organization is assessing it. **Closed** means the SCO has been completely resolved with the concurrence of the CCB members.

The CCB may also assign a **priority and release identifier** to guide the prioritization and organization of concurrent software development activities.

Configuration Baseline

A **configuration baseline** is a named collection of software components and supporting documentation that is placed under change management. It is upgraded, maintained, tested, studied, and eventually obsoleted as the software evolves. With complex configuration management systems, there may be several project-specific and domain-specific standards for baselines.

There are generally **two classes of baselines: external product releases and internal testing releases**. A configuration baseline is a named collection of components that is treated as a unit. It is formally controlled because it acts as a packaged exchange between groups. For example, the development organization may release a configuration baseline to the test organization or even to itself. A project may also release a configuration baseline to the user community for beta testing.

Baseline Release Levels

Generally, three levels of baseline releases are required for most systems: **major, minor, and interim**. Each level corresponds to a numbered identifier such as **N.M.X**, where **N** is the major release number, **M** is the minor release number, and **X** is the interim release identifier.

A **major release** represents a significant generation of the product or project. A **minor release** represents the same basic product with enhanced features, performance, or quality. Major and minor releases are intended to be external product releases that are supported for a period of time. An **interim release** corresponds to a developmental configuration that is intended to be temporary. The figure shows typical release sequences for a large-scale, one-of-a-kind project and a small commercial product. These sequences contain prototype, architecture, internal test, alpha, beta, product, and upgrade releases across the different development phases.

Change Categories

Once software is placed in a controlled baseline, **all changes are tracked**. A distinction must be made between the different causes of change. The changes are classified from **Type 0 to Type 4**.

Type 0 Change

A **Type 0 change** represents critical failures in which defects are nearly always fixed before any external release. These changes generally represent showstoppers that have an impact on the usability of the software in its critical use cases.

Type 1 Change

A **Type 1 change** is a bug or defect that either does not impair the usefulness of the system or can be worked around. Such errors tend to be related to nuisances in critical use cases or serious defects in secondary use cases that have a low probability of occurrence.

Type 2 Change

A **Type 2 change** is an enhancement rather than a response to a defect. Its purpose is to improve performance, testability, usability, or some other aspect of quality that represents good value engineering.

Type 3 Change

A **Type 3 change** is necessitated by an update to the requirements. It may involve new features or capabilities that are outside the scope of the current vision and business case.

Type 4 Change

A **Type 4 change** is a change that is not accommodated by the other categories. Examples include documentation-only changes or version upgrades to commercial components.

The textbook table illustrates these change types in two project environments: a **large-scale air traffic control project** and a **packaged visual modeling tool**. The examples show how the same change categories can represent different types of changes depending on the project domain.

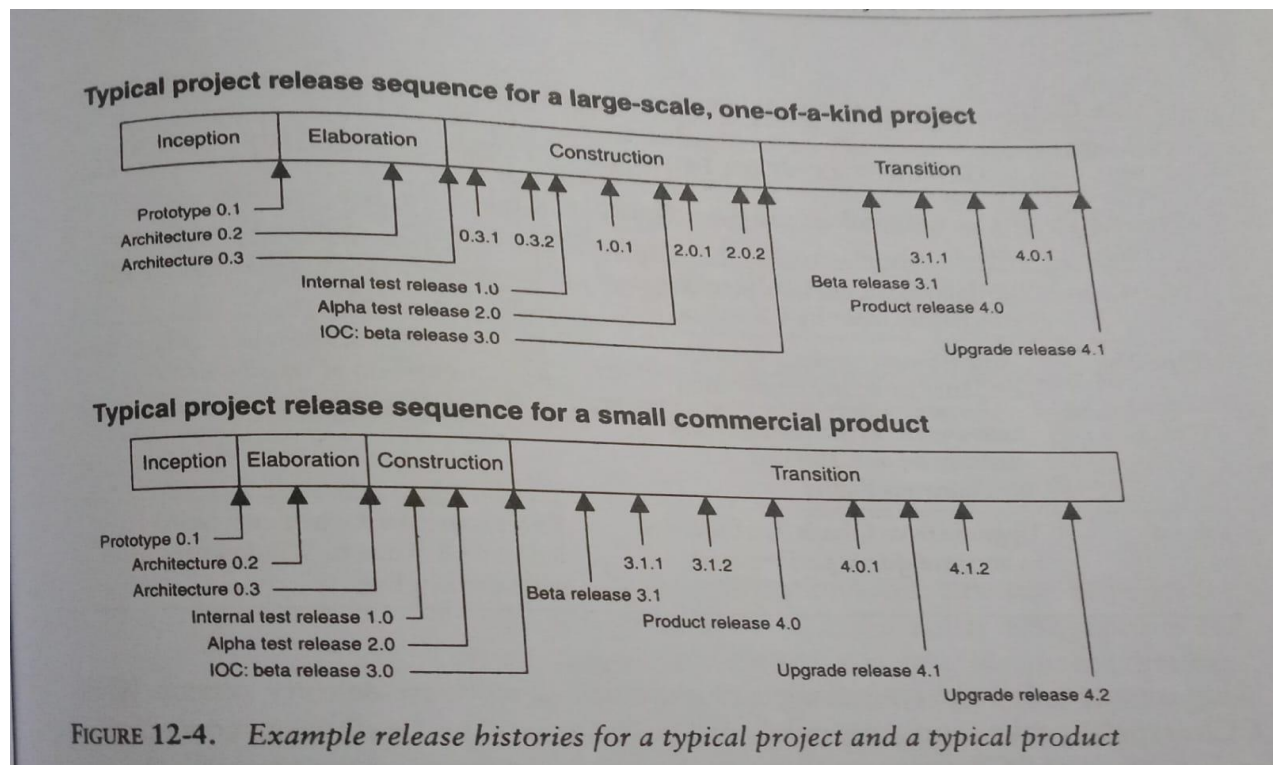


FIGURE 12-4. Example release histories for a typical project and a typical product

Configuration Control Board (CCB):

A **Configuration Control Board (CCB)** is a team of people that functions as the decision-making authority for the content of configuration baselines. A CCB usually includes the software manager, software architecture manager, software development manager, software assessment manager, and other important stakeholders such as the customer, software project manager, systems engineer, and users.

The CCB is responsible for maintaining a controlled software delivery system. CCB actions may be performed through board meetings, online distribution, concurrence, and approval depending on the circumstances of the project.

Software Change Order Process

An iterative development process must include **comprehensive and rigorous change management** of the evolving software baselines. The fundamental process

for controlling software development and maintenance activities can be described through the sequence of states followed by a **Software Change Order (SCO)**.

Proposed

In the **Proposed** state, a proposed change is drafted and submitted to the CCB. The proposed change must contain a technical description of the problem and an estimate of the effort required to resolve it.

Accepted, Archived, or Rejected

In this state, the **CCB assigns a unique identifier** to the proposed change and either accepts, archives, or rejects it. Acceptance means that the change is scheduled for resolution in the next release. Archiving means that the change is accepted but postponed for resolution in a future release. Rejection means that the change is judged to have no merit, to be redundant with other proposed changes, or to be outside the project scope.

Before accepting the SCO, the CCB verifies that all SCO fields are appropriate and accurate. The SCO is then assigned to a responsible person in the development organization for resolution.

In Progress

In the **In Progress** state, the responsible person analyzes, implements, and tests a solution to satisfy the SCO. This work also includes updating documentation, release notes, and SCO metrics, and submitting new SCOs when necessary. After a complete solution is achieved, the responsible person completes the resolution section of the SCO and submits it to the **independent test team** for assessment.

In Assessment

In the **In Assessment** state, the independent test team evaluates whether the SCO has been completely resolved. If the independent test team determines that the change has been satisfactorily resolved, the SCO is submitted to the CCB for final disposition and closure.

Closed

In the **Closed** state, the development organization, independent test organization, and CCB agree that the SCO has been successfully resolved. The SCO is then transitioned to the **closed status**, completing the change-control process.

TABLE 12-1. *Representative examples of changes at opposite ends of the project spectrum*

CHANGE TYPE	AIR TRAFFIC CONTROL PROJECT	PACKAGED VISUAL MODELING TOOL
Type 0	Control deadlock and loss of flight data	Loss of user data
Type 1	Display response time that exceeds the requirement by 0.5 second	Browser expands but does not collapse displayed entries
Type 2	Add internal message field for response time instrumentation	Use of color to differentiate updates from previous version of visual model
Type 3	Increase air traffic management capacity from 1,200 to 2,400 simultaneous flights	Port to new platform such as WinNT
Type 4	Upgrade from Oracle 7 to Oracle 8 to improve query performance	Exception raised when interfacing to MSeExcel 5.0 due to Windows resource management bug

INFRASTRUCTURE:

I.	Process-primitive definitions
A.	Life-cycle phases (inception, elaboration, construction, transition)
B.	Checkpoints (major milestones, minor milestones, status assessments)
C.	Artifacts (requirements, design, implementation, deployment, management sets)
D.	Roles and responsibilities (PRA, SEPA, SEEA, project teams)
II.	Organizational software policies
A.	Work breakdown structure
B.	Software development plan
C.	Baseline change management
D.	Software metrics
E.	Development environment
F.	Evaluation criteria and acceptance criteria
G.	Risk management
H.	Testing and assessment
III.	Waiver policy
IV.	Appendixes
A.	Current process assessment
B.	Software process improvement plan

FIGURE 12-5. *Organization policy outline*

INFRASTRUCTURES

From the **process automation** perspective, an organization's infrastructure provides the capital assets needed for software development.

It mainly contains two key artifacts:

Organization Policy: Defines standards for project software development processes.

Organization Environment: Provides an inventory of tools used to automate the process.

These tools act as **automation building blocks** for configuring project environments efficiently and economically.

Organization Policy

The **organization policy** is usually provided as a handbook. It defines the life cycle and process primitives, including milestones, artifacts, repositories, metrics, roles, and responsibilities.

It provides a framework for answering four questions:

What gets done? — Activities and artifacts.

When does it get done? — Life-cycle phases and milestones.

Who does it? — Team roles and responsibilities.

How do we know that it is adequate? — Checkpoints, metrics, and performance standards.

Balance in Organization Policy

Balance is important while defining organizational policies. Organizations should avoid two extremes: **no institutionalized process** and **excessive standardization and bureaucracy**.

Effective policies should be **concise**. They should avoid unnecessarily lengthy policy documents.

Policies should be limited to the **real requirements** and should be properly enforced.

The word “**should**” should be avoided in policy statements. Policies should define mandatory standards using “**shall**.”

Waivers are exceptions, not the rule. They should be used only when necessary. Policy should also be written at the **appropriate organizational level**.

Levels of Organization Policy

Highest Organization Level

This level focuses on **strategic and long-term process improvements**.

It also supports general technology insertion and education, comparison of project and business-unit performance, and mandatory quality control.

Intermediate Line-of-Business Level

This level focuses on **tactical and short-term process improvement**.

It supports domain-specific technology insertion and training, reuse of components, processes, tools, and personnel experience, and compliance with organizational standards.

Lowest Project Level

This level focuses on achieving **quality, schedule, and cost targets efficiently**.

It also supports project-specific training, customer requirements, and organizational or business-unit standards.

Standardization

Standardization should generally focus on **line-of-business units**, rather than the entire organization or individual projects.

It is most effective at the business-unit level because there is greater commonality and reuse across projects, processes, and tools.

Standardization across different business areas can be difficult because their priorities, tools, techniques, methods, and stakeholder cultures may differ.

Too much standardization can produce weak policies or excessive use of waivers.

Too little standardization allows every project to create its own process and environment. Over time, this can reduce the organization’s ability to achieve **process improvement and growth**.

Organization Policy as a Defining Document

The organization policy is the **defining document for the organization’s software policies**.

During process assessment, it provides evidence of what the organization says it does.

Reviewers can use it to examine projects and personnel and determine whether the organization actually **does what it says**.

Organization Policy Outline

The policy defines **process primitives**, including:

Life-cycle phases: Inception, Elaboration, Construction, and Transition.

Checkpoints: Major milestones, minor milestones, and status assessments.

Artifacts: Requirements, design, implementation, deployment, and management sets.

Roles and responsibilities: PRA, SEPA, SEEA, and project teams.

It also defines organizational software policies for **work breakdown structure, software development plan, baseline change management, software metrics, development environment, evaluation and acceptance criteria, risk management, and testing and assessment**.

The policy also contains a **waiver policy**.

Its appendices include the **current process assessment** and **software process improvement plan**.

ORGANIZATION ENVIRONMENT

The **organization environment** supports automation of the default process.

It provides tools and techniques for automating the process as much as practical.

Standardized Tool Selection

Organizations may invest in site licenses or negotiate discounts with tool vendors.

This encourages project teams to use common tools, promotes common workflows, and improves the return on investment from training.

Standard Notations

Standard notations are used for artifacts.

For example, **UML** can be used for design models, and **Ada 95** can be used for reliability-critical implementation artifacts.

Tool Adjuncts

Tool adjuncts include existing **artifact templates and customizations**. Examples include architecture descriptions, evaluation criteria, release descriptions, and status assessments.

Activity Templates

Activity templates support activities such as **iteration planning, major milestone activities, and configuration control boards**.

Other Useful Infrastructure Components

A **reference library** provides previous experience for planning, assessing, and improving process performance.

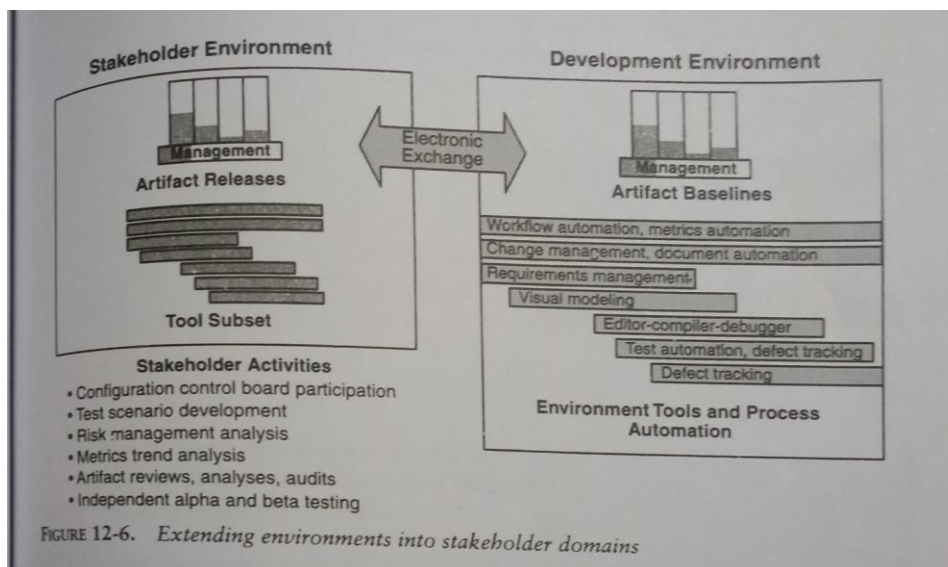
It helps answer questions such as **How well? How much? Why?**

Case studies provide examples and objective performance benchmarks from successful projects that followed the organizational process.

A **project artifact library** contains examples such as software development plans, architecture descriptions, and status assessment histories.

Mock audits and compliance traceability support external process assessment frameworks such as the **Software Engineering Institute's Capability Maturity Model (SEI CMM)**.

STAKEHOLDER ENVIRONMENT:



STAKEHOLDER ENVIRONMENTS

Modern iterative development with automation should not be limited to the development team. Large-scale projects may also involve **external stakeholders**. These stakeholders may include procurement agency contract monitors, end-user engineering personnel, third-party maintenance contractors, independent verification and validation contractors, and regulatory agency representatives.

Access to Development Resources

External stakeholders need access to development environment resources so that they can contribute to the project.

If stakeholders depend only on paper documents, information exchange becomes slow and inefficient.

An **online stakeholder environment** allows them to participate directly in the development process.

They can accept and use **executable increments** for hands-on evaluation. They can also use the same online tools, data, and reports used by the development organization.

This reduces excessive travel, paper exchange delays, format translations, shipping costs, and other overhead costs.

Stakeholder Activities

The stakeholder environment supports activities such as **configuration control board participation, test scenario development, risk management analysis, metrics trend analysis, artifact reviews and audits, and independent alpha and beta testing**.

Extending the Development Environment

The development environment contains **artifact baselines** and management resources. It also provides tools and process automation.

Important facilities include **workflow automation, metrics automation, change management, document automation, requirements management, visual modeling, editor-compiler-debugger, test automation, and defect tracking**.

The stakeholder environment receives **artifact releases** and provides a suitable **tool subset** for stakeholder activities.

Electronic Technical Artifacts

Technical artifacts are not limited to paper documents. **Visual models and source code** can be reviewed more efficiently in electronic form using tools with smart browsers.

Electronic access makes reviews and analysis faster and more effective.

Independent Assessment

Stakeholders can independently assess evolving artifacts through **electronic read-only access** to online data.

This may include **configuration baseline libraries and the change management database**.

Reviews, inspections, breakage/rework assessments, metrics analysis, and beta testing can therefore be performed independently of the development team.

Electronic Delivery of Documents

Even paper documents should be delivered electronically whenever possible.

This helps reduce **production costs and turnaround time**.

Benefits of Shared Environment

When stakeholders can electronically access environment resources, **continuous exchange and feedback** become more efficient and useful.

The development team should provide an open environment with sufficient resources for customer access.

Stakeholders should use this access to **add value to the project without interrupting development**.

Internet and intranet technologies can make such paperless environments economical.

Issues in Stakeholder Environments

Extending development resources to stakeholder domains creates several important issues.

The major issues are **how much access freedom should be provided, who should fund the environment and tools, how secure the information exchange should be, and how change management should be synchronized.**

These issues require changes in the way development teams and stakeholders work together.

PROJECT CONTROL & PROCESS INSTRUMENTATION

INTRODUCTION

Modern software development focuses on managing the major issues involved in developing complex software.

The main themes are architecture-first design, iterative development, component-based development, instrumented change management, and automation.

1. Architecture-First Design

The design should be developed correctly by focusing on the software architecture first.

2. Iterative Development

Risk is managed through iterative development. Development is carried out through repeated iterations so that progress and risks can be controlled.

3. Component-Based Techniques

Software complexity is reduced by using component-based techniques.

4. Instrumented Change Management

Software progress and quality are made measurable through instrumented change management.

5. Automation

Overhead and bookkeeping activities are automated through round-trip engineering and integrated environments.

Software Metrics

It is difficult to manage something that cannot be measured objectively.

In the conventional software process, intermediate products were mainly paper documents. Therefore, measuring software progress and quality was difficult.

Software metrics provide measurements for the activities and products of the software development and integration process.

A software process that depends mainly on manual procedures and human-intensive activities has limited chances of success.

Modern development processes therefore use simple and objective software metrics.

These metrics measure how different aspects of the product and project are changing.

Measuring Progress and Quality

The quality of software products and progress toward project goals must be measurable throughout the software development life cycle.

Software metrics provide the development and management teams with important information.

Assessment of Progress

Metrics provide an accurate assessment of progress to date.

This helps the team understand the actual progress made in the project.

Software Product Quality

Metrics provide insight into the quality of the evolving software product.

They help in understanding how product quality changes during development.

Cost and Schedule Estimation

Metrics provide a basis for estimating the cost and schedule required to complete the product.

The accuracy of these estimates can increase over time as more project information becomes available.

Key Points

The progress toward project goals and the quality of software products must be measurable throughout the software development cycle.

Metrics values provide an important perspective for managing the process.

Metrics trends provide another perspective by showing how measurements change over time.

The most useful metrics are extracted directly from evolving artifacts.

Objective analysis and automated data collection are important for the success of a metrics program.

Subjective assessment and manual data collection are likely to fail.

SEVEN CORE METRICS

Indicators:

An indicator is a metric or a group of metrics that provides an understanding of the software process or software product or a software project. A software engineer assembles measures and produce metrics from which the indicators can be derived. Two types of indicators are:

- (i) Management indicators.
- (ii) Quality indicators.

I.Management Indicators:

The management indicators i.e., technical progress, financial status and staffing progress are used to determine whether a project is on budget and on schedule. The management indicators that indicate financial status are based on earned value system.

- Work and Progress (Work performed over time)
- Budgeted cost and expenditures (Cost incurred over time)
- Staffing and team dynamics (Personnel changes over time)

II.Quality Indicators: The quality indicators are based on the measurement of the changes occurred in software.

- Change traffic and stability (change traffic over time)
- Breakage and modularity (average breakage per change over time)
- Rework and adaptability (average rework per change over time)
- Mean time between failures (MTBF) and maturity (defect rate over time)

THE SEVEN CORE METRICS

Many different metrics are useful for managing a modern software process. The **seven core metrics** are used on software projects.

They are divided into **three management indicators** and **four quality indicators**.

1. Management Indicators

Work and Progress

Work and progress measures the work performed over time.

It is used for iteration planning, plan-versus-actual comparison, and management indication.

The perspectives include SLOC, function points, object points, scenarios, test cases, and SCOs.

Budgeted Cost and Expenditures

This metric measures the cost incurred over time.

It provides financial insight and comparison between planned and actual expenditure.

The perspectives include cost per month, full-time staff per month, and percentage of budget expended.

Staffing and Team Dynamics

This metric measures personnel changes over time.

It is used to compare resource plans with actual resources, hiring rate, and attrition rate.

The perspectives include people added per month and people leaving per month.

2. Quality Indicators

Change Traffic and Stability

This metric measures change traffic over time.

It is useful for iteration planning and monitoring schedule convergence.

It considers SCOs opened and closed by type and by release, component, or subsystem.

Breakage and Modularity

This metric deals with convergence, software scrap, and quality.

It measures average breakage per change over time.

Its perspective includes reworked SLOC per change, classified by type, release, component, or subsystem.

Rework and Adaptability

This metric measures average rework per change over time.

It is used for convergence, software rework, and quality indication.

Its main perspective is average hours per change, by type, release, component, or subsystem.

MTBF and Maturity

MTBF means Mean Time Between Failures.

It measures failure rate over time and indicates software maturity.

It is related to test coverage, robustness for use, and quality.

Its perspectives include failure counts and test hours until failure, by release, component, or subsystem.

Two Dimensions of Metrics

Each metric has two dimensions: a static value and a dynamic trend.

The static value provides an objective, while the dynamic trend helps to manage the achievement of that objective.

Metrics provide insight into the process, while metrics trends over time show how the process and product are developing.

In iterative development, the main focus is on managing and measuring change.

Therefore, productivity and quality values are secondary until the main management goal is achieved: predictable cost and schedule performance for a given level of quality.

Use of Seven Core Metrics

The seven core metrics can help in managing projects and organizations.

Historical metric values from earlier iterations and projects can be used for planning future iterations and projects.

Once metric collection becomes established, the organization can improve its ability to predict cost, schedule, and quality performance.

Characteristics of the Seven Core Metrics

The seven core metrics are based on common sense and practical experience from successful and unsuccessful metrics programs.

They are simple, objective, easy to collect, easy to interpret, and hard to misinterpret.

Metric collection can be automated and nonintrusive.

They provide consistent assessments throughout the life cycle and are derived from evolving product baselines rather than subjective assessment.

They are useful to both management and engineering personnel for communicating progress and quality in a consistent format.

Their fidelity improves across the life cycle.

Metrics Across the Life Cycle

Metrics used during the engineering stage are less accurate because this stage involves intellectual freedom and risk resolution.

Metrics used during the production stage are more useful because this stage involves implementation activities and change management.

Therefore, the metrics are tailored to the production stage, where cost risk is high and management value is important.

During the engineering stage, metric activities mainly focus on establishing initial baselines and expectations for the production-stage plan.

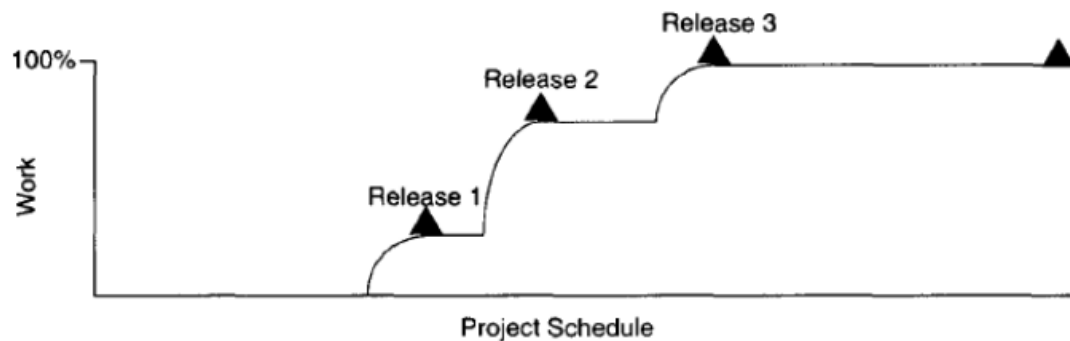
Overview of the seven core metrics

METRIC	PURPOSE	PERSPECTIVES
Work and progress	Iteration planning, plan vs. actuals, management indicator	SLOC, function points, object points, scenarios, test cases, SCOs
Budgeted cost and expenditures	Financial insight, plan vs. actuals, management indicator	Cost per month, full-time staff per month, percentage of budget expended
Staffing and team dynamics	Resource plan vs. actuals, hiring rate, attrition rate	People per month added, people per month leaving
Change traffic and stability	Iteration planning, management indicator of schedule convergence	SCOs opened vs. SCOs closed, by type (0,1,2,3,4), by release/component/subsystem
Breakage and modularity	Convergence, software scrap, quality indicator	Reworked SLOC per change, by type (0,1,2,3,4), by release/component/subsystem
Rework and adaptability	Convergence, software rework, quality indicator	Average hours per change, by type (0,1,2,3,4), by release/component/subsystem
MTBF and maturity	Test coverage/adequacy, robustness for use, quality indicator	Failure counts, test hours until failure, by release/component/subsystem

MANAGEMENT INDICATORS:

There are three fundamental sets of management metrics; technical progress, financial status staffing progress. By examining these perspectives, management can generally assess whether a project is on budget and on schedule. The management indicators recommended here include standard financial status based on an earned value system, objective technical progress metrics tailored to the primary measurement criteria for each major team of the organization and staff metrics that provide insight into team dynamics.

WORK & PROGRESS:



Expected progress for a typical project with three major releases

WORK AND PROGRESS

In an iterative development project, work can be measured by first defining a planned estimate of the work using an objective measure.

The actual progress, or work completed over time, is then compared with the planned work.

Each major organizational team should have at least one primary progress perspective against which its work is measured.

Expected Project Progress

A typical project may have three major releases:

Release 1 → Release 2 → Release 3

As each release is completed, the amount of completed work increases. The project eventually approaches 100% work completion.

Progress Measures for Teams

Software Architecture Team:

Progress is measured by use cases demonstrated.

Software Development Team:

Progress is measured using SLOC under baseline change management and SCOs closed.

Software Assessment Team:

Progress is measured using SCOs opened, test hours executed, and evaluation criteria met.

Software Management Team:

Progress is measured by milestones completed.

Key Idea

Work and progress are measured by comparing planned work with actual work completed over time using suitable measures for each team.

Budgeted Cost and Expenditures:

Need for Cost Measurement

To maintain management control, cost expenditures must be measured throughout the project life cycle.

Metrics for work and progress help compare technical progress with cost expenditures.

In an iterative development process, near-term activities should be planned in detail. Long-term activities can remain as rough estimates and can be refined during later iterations.

Financial Performance Measurement

Financial progress can be measured using an earned value system.

It provides detailed information about cost and schedule performance.

The major difficulty in software projects is measuring the percentage of technical progress objectively and accurately.

Earned value is especially useful during the production stage, where actual progress can be tracked more accurately.

Parameters of Earned Value System

1. Expenditure Plan

The expenditure plan is the planned spending profile of a project over its planned schedule.

For software projects, it generally follows the staffing profile.

2. Actual Progress

Actual progress represents the technical accomplishment compared with the planned progress.

In a healthy project, actual progress should closely follow planned progress.

3. Actual Cost

Actual cost is the actual spending profile of the project over its actual schedule.

In a healthy project, it should closely follow the planned cost profile.

4. Earned Value

Earned value represents the planned cost of the actual progress achieved.

It indicates the value of the work that has actually been completed.

5. Cost Variance

Cost variance is the difference between actual cost and earned value.

A positive value indicates an over-budget situation.

A negative value indicates an under-budget situation.

6. Schedule Variance

Schedule variance is the difference between planned cost and earned value.

A positive value indicates a behind-schedule situation.

A negative value indicates an ahead-of-schedule situation.

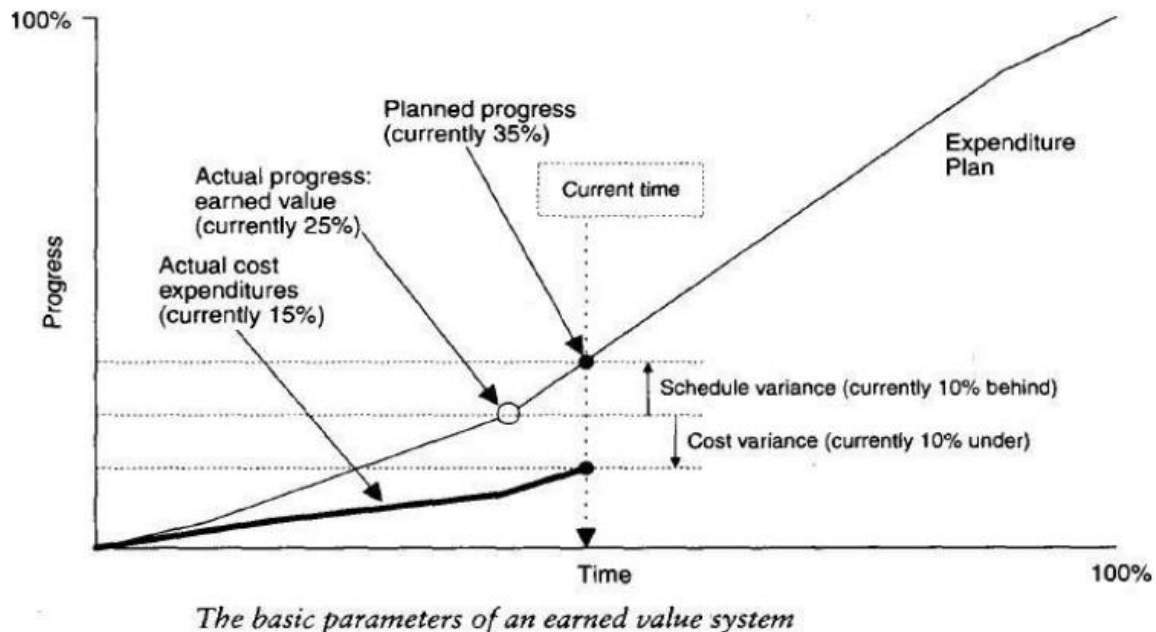
Purpose of Core Metrics

The main purpose of these metrics is to help management and engineering teams assess actual project progress objectively and accurately.

Among the earned value parameters, actual progress is the most subjective.

Cost and schedule usage can be measured more easily, but determining the actual technical progress is difficult.

Therefore, accurate assessment of actual progress is important for understanding project health.



Example of Book Development

The textbook uses **book development** as an example for explaining earned value measurement.

Each part of the book was assigned a **state and earned value**, representing its percentage of completion.

Progress States

0–50%: Content is incomplete.

50%: First draft text and artwork are completed.

65%: Initial text baseline and initial text editing are complete.

75%: Reviewable baseline is available and text and art editing are complete.

80%: Updated baseline is available and cross-chapter consistency is checked.

90%: Reviewed baseline is available and external reviewer comments are incorporated.

100%: Final edit is completed and the editor completes the final cleanup pass.

Actual Progress Assessment

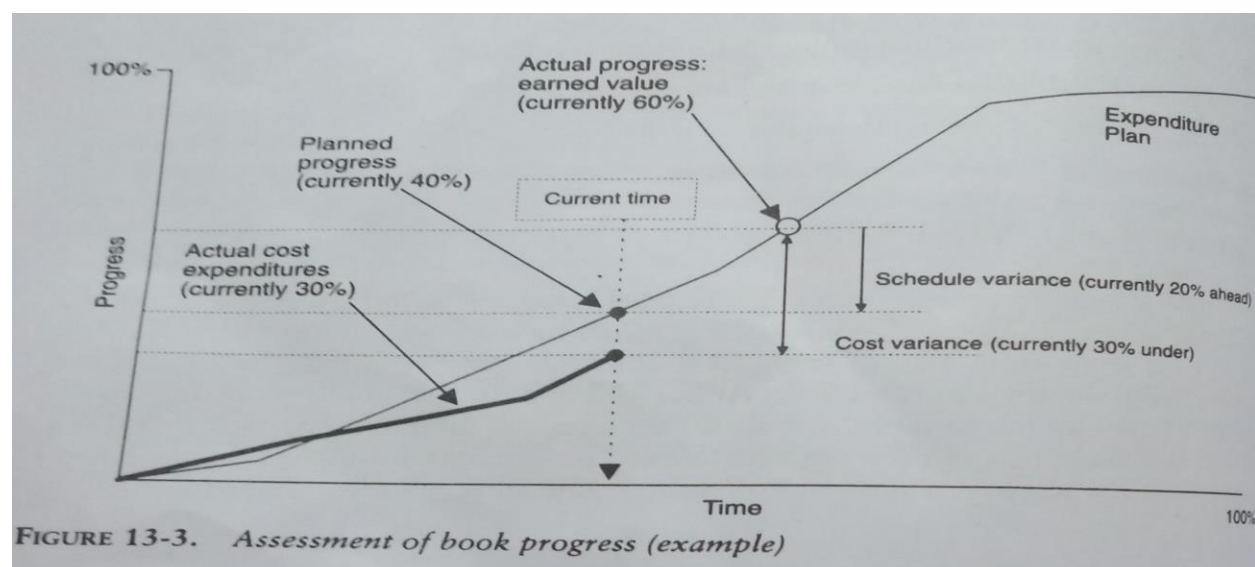
The percentage-complete values were assigned based on experience in writing complex documents.

The planned effort was **10 months** with an expenditure of **\$10,000** for supporting labor.

At month 4, about **400 pages out of 425 pages** had been drafted.

After considering the completion level of each part, the **weighted-average progress was 60%**.

TABLE 13-2. Measurement of actual progress of book development (example)			
COMPONENT	PAGES	%	STATUS
Part I	60	75%	Reviewable baseline
Part II	75	75%	Reviewable baseline
Part III	90	65%	Initial text baseline
Part IV	30	65%	Initial text baseline
Part V	130	50%	Draft content
Other (preface, glossary, index)	40	25%	Content incomplete



Measurement of Actual Progress

The book was divided into several components, and each component was assigned a percentage complete based on its current status.

Component Pages % Status

Part I	60	75% Reviewable baseline
Part II	75	75% Reviewable baseline
Part III	90	65% Initial text baseline
Part IV	30	65% Initial text baseline
Part V	130	50% Draft content
Other	40	25% Content incomplete

At month 4, the assessment showed **20% ahead of schedule** and **30% under budget**. Figure 13-3 represents this assessment graphically. The planned progress was **40%**, actual progress or earned value was **60%**, and actual cost expenditure was **30%**.

Thus, the difference between actual progress and planned progress represents the **schedule variance**, while the difference between earned value and actual expenditure represents the **cost variance**.

Attributes of Effective Progress Assessment

This example provides a framework for understanding important attributes of planning and actual progress assessment. These include establishing an **objective basis**, developing a suitable **work breakdown structure**, and planning with appropriate **fidelity**.

Objective Criteria for Progress

Objective criteria should be established for determining the percentage complete of each component. In the book example, these criteria were based on the author's

experience, personal development style, and understanding of the relationship with the technical editor.

For software projects, the criteria should depend on factors such as the **culture of the team, experience of the team, and development style**. These factors should guide how progress is assessed objectively.

Work Breakdown Structure

A **work breakdown structure (WBS)** divides the project work into suitable parts.

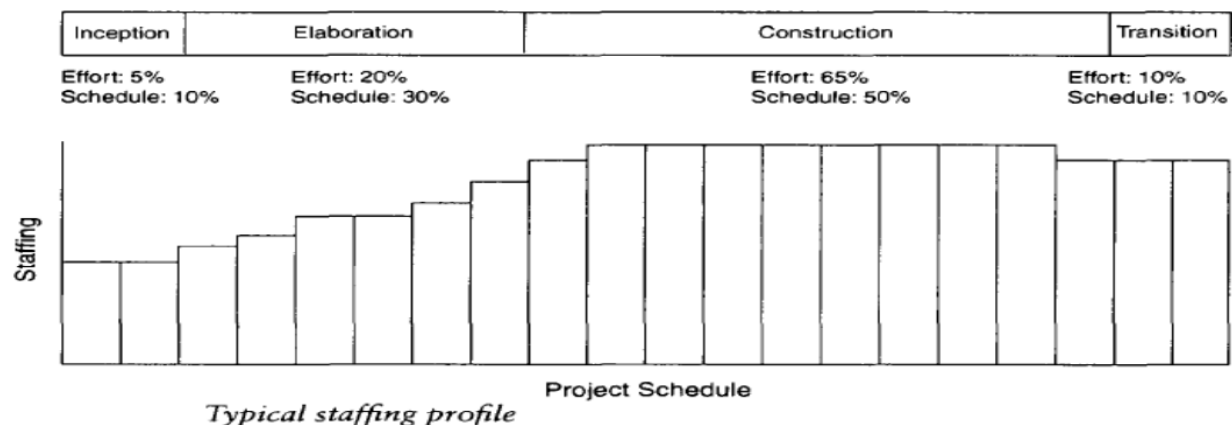
In the book example, the work was divided into different parts or groups of chapters.

However, frequent changes in the book architecture would make detailed component tracking difficult and could introduce unnecessary rework.

Therefore, a better approach is to organize the WBS for tracking the **overall project progress**.

The contributions of the **author, editor, artists, reviewers, and publisher** can be included, while only the construction progress needs to be tracked explicitly by component.

Staffing and Team Dynamics:



Staffing and Team Dynamics

Staffing in Iterative Development

An iterative development project should start with a small team. The team can be increased after the risks in requirements and architecture are suitably resolved.

Staffing can vary depending on the overlap of iterations and other project-specific conditions.

For a one-time development project, the maintenance team is normally smaller than the development team.

For a commercial product, the maintenance and development teams may be of the same size.

For a long-lived product, maintenance becomes a continuous process of constructing new and improved releases.

Typical Staffing Profile

The staffing profile shown in Figure 13-4 follows four phases:

Inception → Elaboration → Construction → Transition

Phase	Effort Schedule	
Inception	5%	10%
Elaboration	20%	30%
Construction	65%	50%
Transition	10%	10%

Staffing generally **increases during the early phases**, reaches its highest level during **Construction**, and decreases during **Transition**.

Tracking Staffing

Actual staffing versus planned staffing is an important management metric.

It helps managers understand whether the project is following the planned staffing pattern.

Attrition and Additions

The relationship between staff additions and attrition is another important indicator of project momentum.

Adding new staff can sometimes slow project progress because existing team members must spend productive time helping new people understand the project.

Importance of Low Attrition

Low attrition of good people is a sign of success.

Engineers are motivated when they see progress in getting the software to work. This motivation is an important part of an efficient iterative development process.

If this motivation is missing, good engineers may leave the project.

Unplanned Attrition

Unplanned attrition means people leaving a project prematurely.

An increase in unplanned attrition is a serious indicator of project trouble.

The common causes include dissatisfaction with management methods, lack of teamwork, and the possibility of failure in meeting planned objectives.

II.QUALITY INDICATORS:

The four quality indicators are based primarily on the measurement of software change across evolving baselines of engineering data (such as design models and source code).

Change Traffic and Stability:

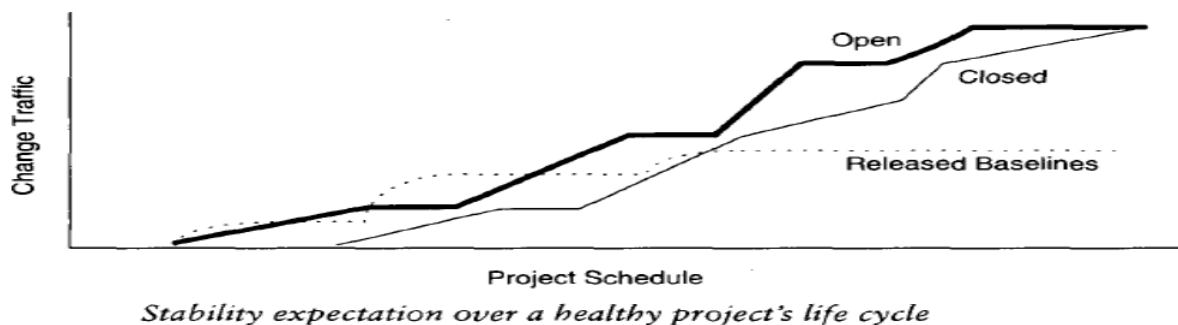
Change traffic is the number of software change orders opened and closed during the software life cycle. It is an indicator of project progress and quality.

Change traffic can be measured according to change type, release, all releases, team, component, subsystem, and other categories. When combined with work and progress metrics, it gives information about the stability of the software.

Stability is the relationship between opened SCOs and closed SCOs. SCO means Software Change Order. When the number of opened and closed changes becomes balanced, the software moves toward stability.

Change traffic compared with the release schedule gives information about schedule predictability. It indicates how well the software development process is performing.

In a healthy project, the change traffic should show a movement toward stability as the project progresses.



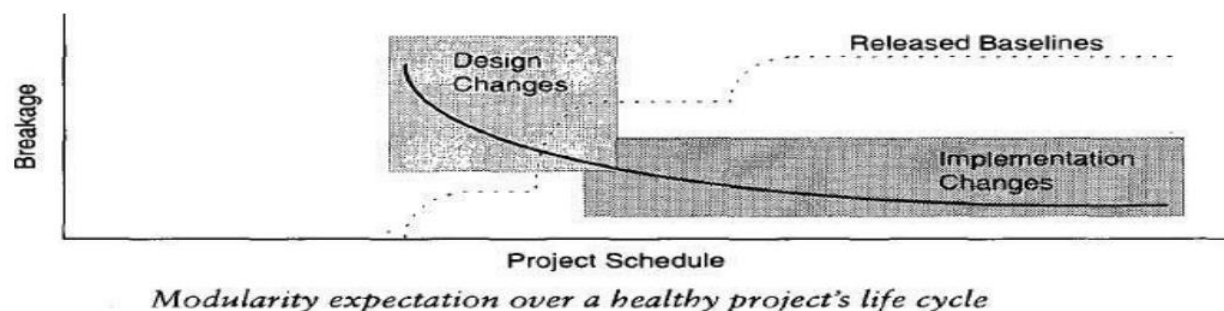
Breakage and Modularity:

Breakage is the average extent of change. It represents the amount of a software baseline that needs rework, measured in terms such as SLOC, function points, components, subsystems, and files.

Modularity is the average breakage trend over time. For a healthy project, the breakage trend is expected to be decreasing or stable.

This indicator provides information about whether software changes are benign or problematic. In a mature iterative development process, earlier changes are expected to produce more scrap than later changes.

If the breakage trend increases with time, it indicates that the maintainability of the product may be suspect.



Rework and Adaptability:

Rework is the average cost of change. It is the effort required to analyze, resolve, and retest all changes made to software baselines.

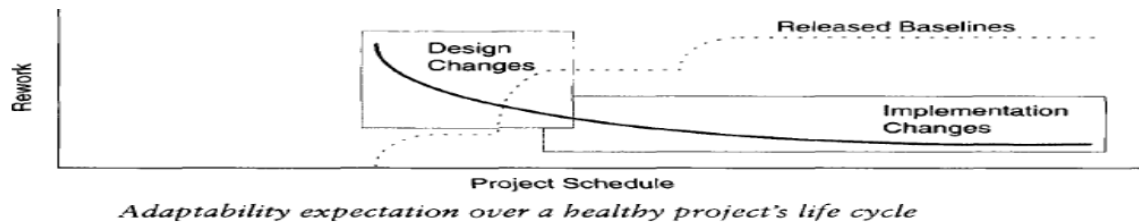
Adaptability is the rework trend over time. For a healthy project, the expected trend is decreasing or stable.

Not all software changes require the same amount of effort. Some changes can be completed in a staff-hour, while others may require staff-weeks. Therefore, this metric provides useful information about the amount of rework required for changes.

In a mature iterative development process, earlier changes, such as architectural changes, are expected to require more rework because they can affect multiple components and people.

Later implementation changes generally require less rework because they tend to be limited to a single component or person.

If the rework trend increases with time, it clearly indicates that the maintainability of the product is suspect.



MTBF and Maturity:

MTBF (Mean Time Between Failures) is the average usage time between software faults. In simple terms, MTBF is calculated by dividing the test hours by the number of Type 0 and Type 1 SCOs.

Maturity is defined as the MTBF trend over time. A healthy project is expected to show an improvement in MTBF as the software becomes more mature.

Early understanding of maturity requires an effective test infrastructure. Traditional testing of monolithic software focused on achieving complete test coverage of every line of code and every change.

For modern distributed and componentized software systems, complete test coverage is practical mainly for individual components. Systems containing many components are tested more effectively using statistical techniques. Therefore, maturity metrics measure statistics over usage time rather than only product coverage.

Software Error Types

Software errors can be classified into two types: deterministic errors and nondeterministic errors.

Bohr-bugs are deterministic errors. They occur whenever the software is stimulated in a particular way. They are mainly caused by coding errors, and the required changes are usually isolated to a single component.

Heisen-bugs are nondeterministic software faults. They occur with a certain probability under a particular situation. They are usually associated with design errors, may require changes in multiple components, and may not be repeatable even when the software is executed in apparently the same way.

To obtain adequate test coverage and identify significant Heisen-bugs, extensive statistical testing is required using realistic and randomized usage scenarios.

Traditional software running as a single program on a single processor generally contained mainly Bohr-bugs. Modern distributed systems with many interacting components across a network are more vulnerable to Heisen-bugs, which are more difficult to detect, analyze, and resolve.

Testing for Software Maturity

An effective way to mature a software product is to establish an initial test infrastructure early in the life cycle. This infrastructure should support randomized usage scenarios and continuously increase the breadth and depth of usage scenarios.

Once baselines are established, they should be continuously subjected to test scenarios. Reliability metrics can then be obtained from this testing.

Product maturity can be understood by maximizing test time through independent test environments, automated regression testing, randomized statistical testing, and after-hours stress testing.

This testing approach encourages test automation early in the life cycle. It can also be used to monitor performance improvements and measure software reliability.



LIFE CYCLE EXPECTATIONS

There is no mathematical or formal derivation for using the seven core metrics. However, there are specific reasons for selecting these metrics.

Reasons for Selecting the Seven Core Metrics

The quality indicators are derived from the evolving product rather than from the artifacts. This means that the metrics focus on how the product develops and changes during the project life cycle.

The metrics also provide insight into the waste generated by the process. Scrap and rework metrics are standard measurements used in many manufacturing processes. Therefore, these metrics help in understanding the amount of unnecessary work or waste produced during development.

These metrics recognize the inherently dynamic nature of an iterative development process. Instead of concentrating only on the current value, they explicitly focus on the trends or changes with respect to time.

The combination of information from the current value and the current trend provides tangible indicators for management action. Thus, management can use both the present condition and its direction of change to take appropriate action.

Variation of Metric Values

The actual values of these metrics can vary widely across projects, organizations, and domains. However, the relative trends across the different project phases should generally follow the pattern shown in Table 13-3.

A mature development organization should be able to define metric targets that are much more definitive and precise for its particular line of business and specific development processes.

The default pattern of life-cycle metrics evolution

METRIC	INCEPTION	ELABORATION	CONSTRUCTION	TRANSITION
Progress	5%	25%	90%	100%
Architecture	30%	90%	100%	100%
Applications	<5%	20%	85%	100%
Expenditures	Low	Moderate	High	High
Effort	5%	25%	90%	100%
Schedule	10%	40%	90%	100%
Staffing	Small team	Ramp up	Steady	Varying
Stability	Volatile	Moderate	Moderate	Stable
Architecture	Volatile	Moderate	Stable	Stable
Applications	Volatile	Volatile	Moderate	Stable
Modularity	50%–100%	25%–50%	<25%	5%–10%
Architecture	>50%	>50%	<15%	<5%
Applications	>80%	>80%	<25%	<10%
Adaptability	Varying	Varying	Benign	Benign
Architecture	Varying	Moderate	Benign	Benign
Applications	Varying	Varying	Moderate	Benign
Maturity	Prototype	Fragile	Usable	Robust
Architecture	Prototype	Usable	Robust	Robust
Applications	Prototype	Fragile	Usable	Robust

Overall Life-Cycle Pattern

During Inception, the project is at an early stage. Progress and effort are low, staffing is a small team, stability is volatile, and the product is mainly at the prototype level.

During Elaboration, the project develops its architecture significantly. Effort and schedule increase, staffing begins to ramp up, and stability becomes more moderate. The architecture becomes usable, while applications can still be fragile.

During Construction, most of the development work is completed. Progress, effort, and schedule reach high levels, staffing becomes steady, and stability improves. The architecture becomes stable and the applications become usable.

During Transition, the project reaches 100% progress, effort, and schedule. The architecture and applications become stable, adaptability becomes benign, and the overall product reaches a robust level of maturity.

PRAGMATIC SOFTWARE METRICS

Introduction

Measurement is useful for decision making. It provides data that helps decision makers ask the right questions, understand the situation, and make objective decisions.

Software projects are highly dynamic. Therefore, metrics should be available at any time and should be adaptable to different parts of the product, such as release, version, component, and class.

Metrics should also be maintained over time so that trends can be studied. In practice, this is best achieved when metrics are maintained online as an automated by-product of the development and integration environment.

Characteristics of a Good Metric

1. Meaningful to Stakeholders

A good metric should be meaningful to the customer, manager, and performer. If any one of these stakeholders does not find the metric meaningful, it will probably not be used.

Customers accept metrics when they are shown to be meaningful to the developer.

2. Quantifiable Correlation with Business Performance

A good metric should show a quantifiable relationship between process changes and business performance.

The main organizational objectives are financial objectives such as:
Cost reduction, revenue increase, and margin increase.

Therefore, metrics should help relate software process changes to business performance.

3. Objective and Unambiguously Defined

A metric should be objective and clearly defined.

It should preferably use numerical representations such as:

Numbers, percentages, and ratios.

Subjective terms such as *excellent*, *good*, *fair*, and *poor* should be avoided.

Ambiguity can be reduced by using well-defined units such as staff-month, SLOC, change, function point, class, scenario, and requirement.

4. Displays Trends

A good metric should display trends.

It is important to understand how the value of a metric changes with respect to time, subsequent projects, and subsequent releases.

This is especially important in iterative development models.

A metric normally provides a perspective. The manager, team, or other decision authority must interpret the metric and decide what action is required.

5. Natural By-Product of the Process

A good metric should be a natural by-product of the process.

It should not introduce new artifacts or additional overhead activities.

Instead, it should be obtained directly from the normal software engineering and management workflows.

6. Supported by Automation

A good metric should be supported by automation.

Experience shows that successful metrics are usually collected and reported by automated tools.

Software tools also require rigorous definitions of the data they process, which supports consistent measurement.

INTERPRETING METRICS

When a metric shows a problem, it is important to identify the actual cause behind the symptoms.

Metrics usually show the effects. Finding the causes requires combining different perspectives and applying proper reasoning.

For example, a low number of change requests may mean that the software is mature and error-free. It may also mean that the test team is on vacation.

A software change order open for a long time may mean that substantial rework is required. It may also mean that diagnosis took a long time, while the final solution required only a simple one-line change.

A large increase in personnel may increase progress if the new people are trained and productive. However, progress may decrease if they are untrained and require extensive support.

Final Point

Metrics provide information and perspectives, but they do not make value judgments.

The interpretation of metrics and the decision about necessary actions must be left to knowledgeable people such as software project managers.

METRICS AUTOMATION

INTRODUCTION

There are many opportunities to automate the project control activities of a software project. For managing a project against a plan, a Software Project Control Panel (SPCP) maintains an online view of the current status of evolving project artifacts.

The SPCP is similar to a project “dashboard.” It integrates data from multiple sources and displays the current status of different aspects of the project. A project manager may view overall project values, a test manager may focus on metrics for an upcoming release, and a development manager may view information about particular subsystems or components.

The panel can use warning lights, thresholds, variable scales, digital formats, and analog formats to provide an overview of the current situation. It can also support detailed analysis of the project. This automation improves management understanding of progress and quality trends and increases the acceptance of metrics by the engineering team.

COMPONENTS OF A COMPLETE SPCP

To implement a complete SPCP, several components are required.

Metrics primitives provide basic forms of metrics such as indicators, trends, comparisons, and progressions.

A graphical user interface (GUI) provides support for the software project manager and can also be adapted for other roles.

Metrics collection agents extract data from environment tools that maintain engineering notations for different artifact sets.

A metrics data management server stores the data extracted by the agents and provides the data required to populate the metric displays of the GUI.

Metrics definitions specify the actual metric presentations. These include requirements progress, design progress, implementation progress, assessment progress, and other progress dimensions. The data may be extracted from requirements, design, implementation, and deployment artifacts, as well as manual sources and financial management systems.

Actors are normally the monitor and administrator.

ROLES IN METRICS AUTOMATION

Specific monitors, called roles, may include software project managers, software development team leads, software architects, and customers. Each role can have a different panel configuration and data scope, although the general use cases remain similar.

The Monitor defines panel layouts using existing mechanisms, graphical objects, and links to project data. The monitor can also query data for display at different levels of abstraction.

The Administrator installs the system and defines new mechanisms, graphical objects, and linkages. The administrator also handles archiving and defines composition and decomposition structures for displaying multiple levels of information.

Panels and Graphical Objects

The complete display is called a panel. A panel contains graphical objects, such as dials and bar charts, used to display information.

Each graphical object displays a metric. The metric is labelled with its metric type, summary level, and instance name, such as lines of code, subsystem, or server.

Metrics can be displayed in two modes: value and graph. A value represents the metric at a particular point in time, while a graph represents multiple and consecutive points in time.

Control Values

Metrics may be displayed with or without control values. A control value is an existing expectation, either absolute or relative, used for comparison with a dynamically changing metric.

For example, the planned value for a progress metric can be used as a control value to compare with the actual value. Thresholds are another type of control value. When a threshold is crossed, the system can change its state and make the condition visible to the user.

Control values can be displayed together with the corresponding metric in the same graphical object, allowing easy visual comparison.

Indicators

An indicator displays data in different formats. These may be binary, tertiary, digital, or other enumerated formats.

For example, an indicator may use different states such as red, yellow, and green. Indicators can summarize a condition associated with another metric or show the relationship between a metric and its control values.

Trend, Comparison, and Progression

A trend graph presents metric values over time. It allows upper and lower thresholds to be defined. When a threshold is crossed, an associated indicator can show a noticeable change in state.

Trends can use different time intervals such as day, week, month, quarter, or year.

A comparison graph presents multiple metric values together over time. It can show whether the values are converging or diverging.

A progression graph presents the percentage complete of a project or activity. Different elements of progress are shown as transitions between states, and an earned value can be associated with each state.

These three fundamental metric classes are illustrated in Figure 13-9.

Summary and Drill-Down

Metric information can be summarized using a user-defined linear structure. For example, lines of code can be summarized by unit, subsystem, and project.

The project is the top-level qualifier for all data belonging to a particular set. Users can define summary structures for lower levels and select the required display level.

The user can also drill down from a summarized value to view the details at lower levels.

Example SPCP Display

Figure 13-10 shows a simple SPCP for a project. The software project manager has a top-level display containing four graphical objects.

Project Activity Status

The upper-left graphical object provides an overview of the status of the top-level WBS elements. The elements can be represented using red, yellow, and green states according to their current earned-value status.

For example, green represents ahead of plan, yellow represents within 10% of plan, and red represents elements having more than 10% cost or schedule variance.

This graphical object can show several indicators, including tertiary colors, actual percentage, and the current first derivative. An upward arrow indicates that the situation is getting better, while a downward arrow indicates that it is getting worse.

Technical Artifact Status

The upper-right graphical object provides an overview of the status of the evolving technical artifacts.

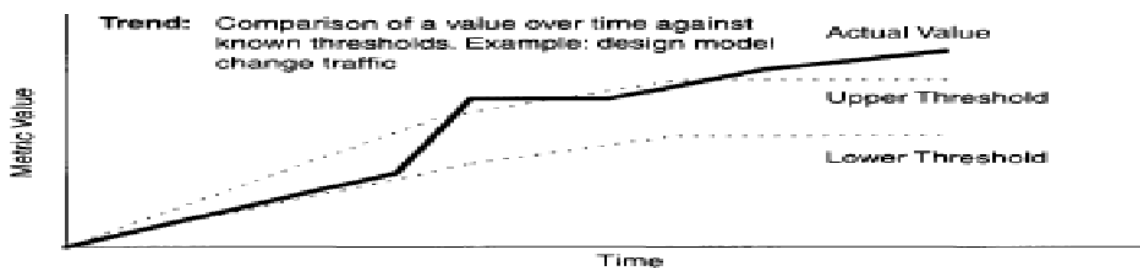
The Req light represents the current state of use-case models and requirements specifications. The Des light represents design models. The Imp light represents the source-code baseline. The Dep light represents the test program.

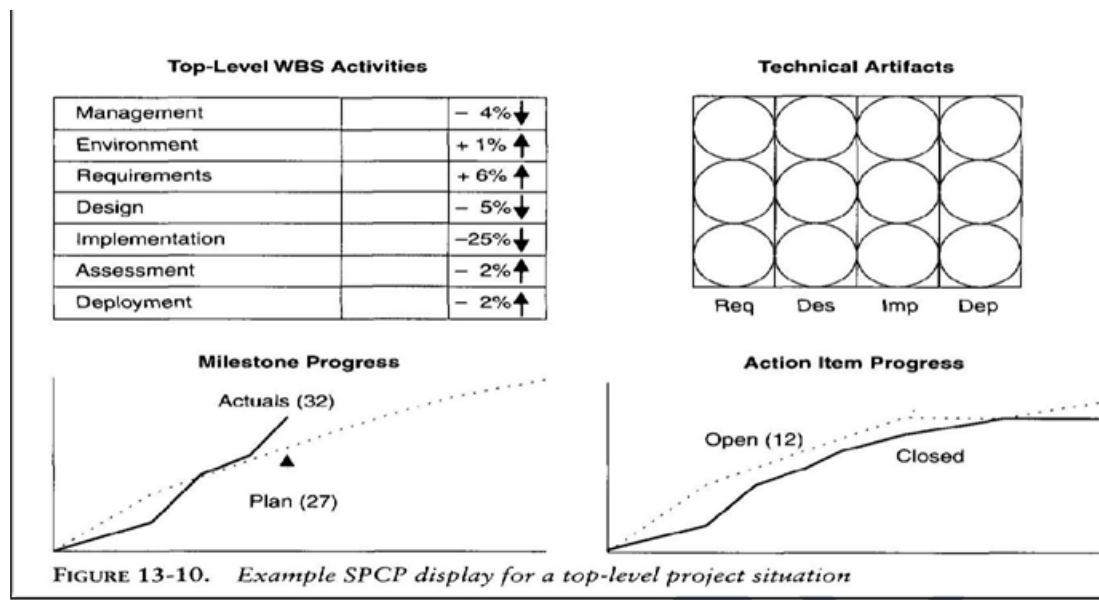
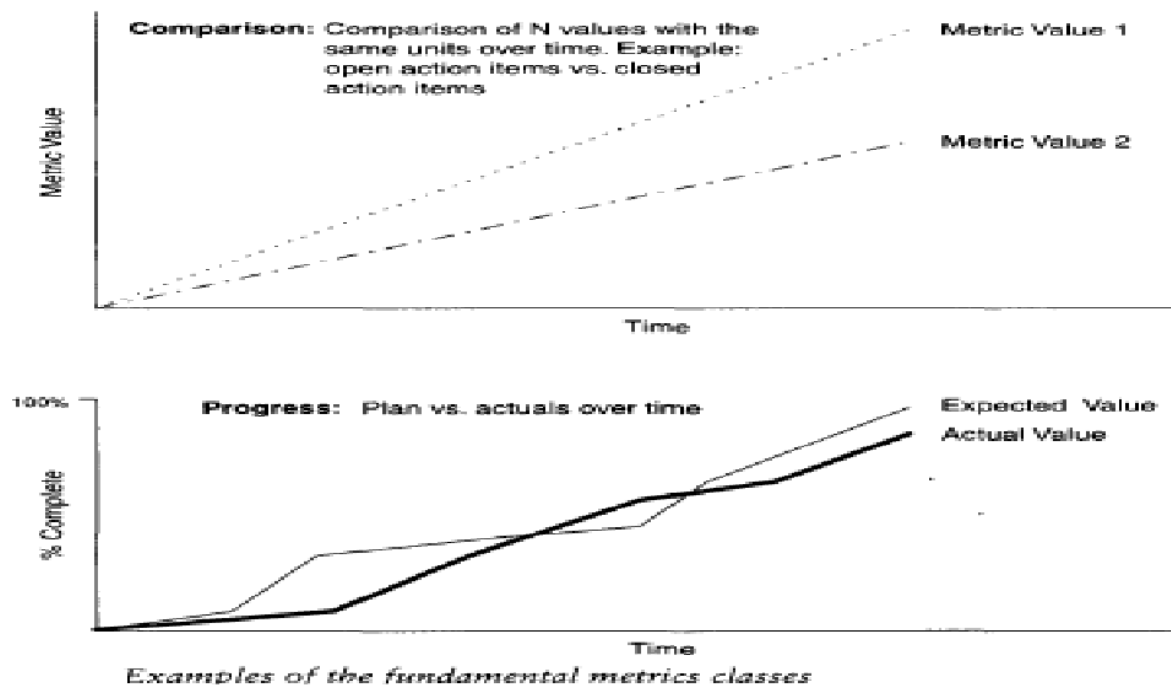
Milestone Progress

The lower-left graphical object provides a progress assessment of milestones against the plan. It also provides indicators showing the current values.

Action Item Progress

The lower-right graphical object provides another view of project progress. It shows the current number of open and closed action items.





CONFIGURABLE SPCP DISPLAY

Figure 13-10 is an example of a progress metric implementation. The **format and content of a project panel are configurable** according to the software project manager's requirements.

Some managers may want only summary data and a few important trends. Others may require many trends and detailed information.

An SPCP should support **tailoring and drill-down capability**. For example, selecting a red indicator for deployment artifacts can provide more detailed information, either over time through a trend chart or at a lower abstraction level such as test status for each release or subsystem.

Top-Level SPCP Use Cases

The basic operations of an SPCP can be understood through the following activities.

Start the SPCP: The SPCP starts and displays the most recent information that was saved when the user last used the system.

Select a panel preference: The user selects a previously defined panel preference, and the SPCP displays the selected configuration.

Select a value or graph metric: The user chooses whether the metric should be displayed as a value for a particular point in time or as a trend. The default value is the most recent measurement, while the default trend is monthly.

Select to superimpose controls: The user selects a graphical object and requests that its control values be displayed. For trends, the control values are shown together with the metric.

Drill down to trend: The user selects a graphical object showing a point in time and drills down to view the trend for that metric.

Drill down to point in time: The user selects a graphical object displaying a trend and drills down to view the metric value at a particular point in time.

Drill down to lower levels of information: The user selects a graphical object and moves to the next lower level of information.

Drill down to lower levels of indicators: The user selects an indicator and views the detailed breakdown of indicators at the next level.

