UNIT-VI
PART - A

Process Automation: Tools: Automation Building Blocks, The Project Environment.

Tools : Automation Building Blocks

- Many tools are available to automate the software development process.
- Most of the core software development tools map closely to one of the process workflows, as illustrated in below figure.
- Each of the process workflows has a distinct need for automation support. In some cases, it is necessary to generate an artifact; in others, it is needed for simple bookkeeping.

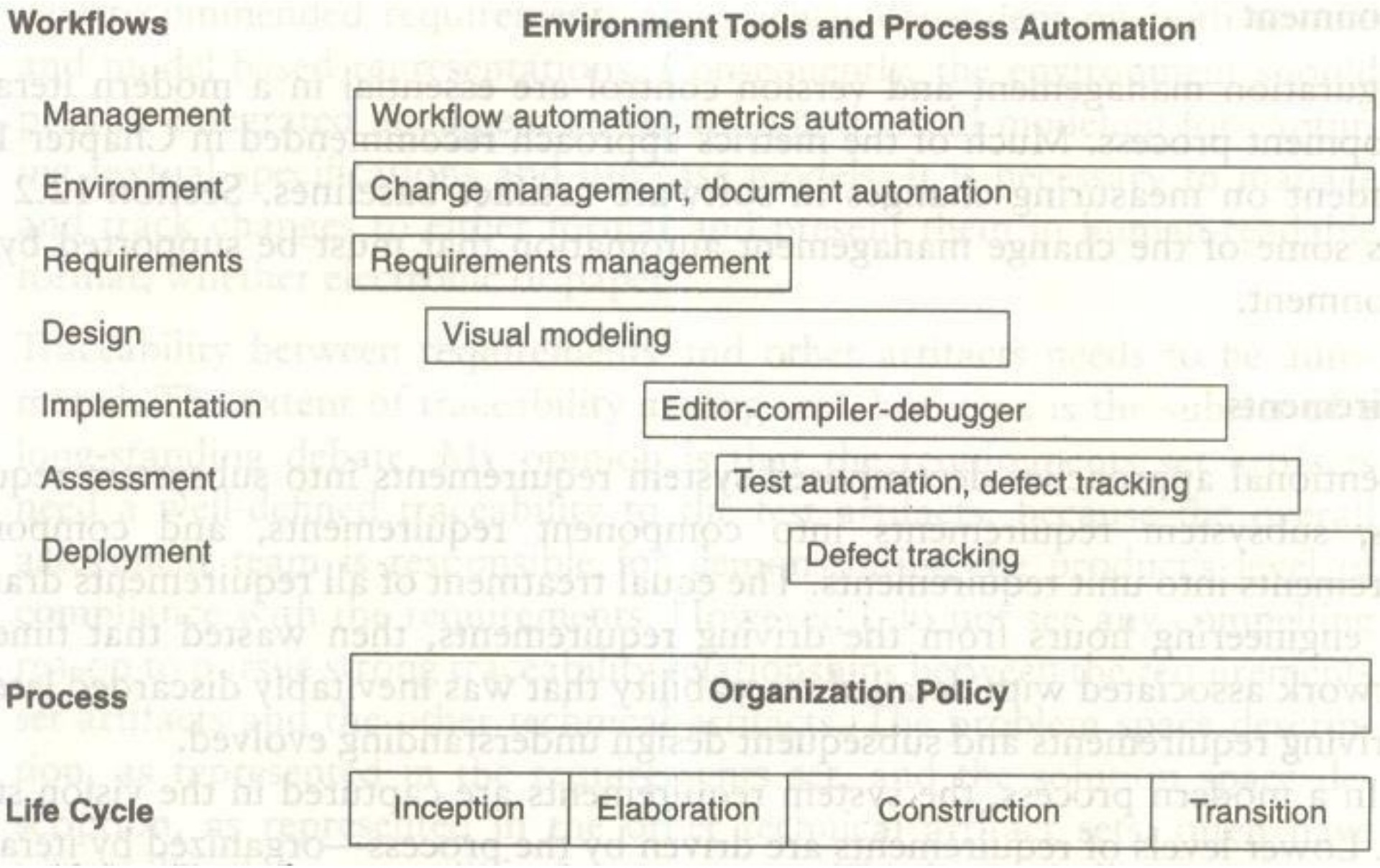


Figure: Typical automation and tool components that support the process workflows

Management

- There are many opportunities for automating the project planning and control activities of the management workflow.
- Software cost estimation tools and WBS tools are useful for generating the planning artifacts.
- For managing against a plan, workflow management tools and a software project control panel that can maintain an on-line version of the status assessment are advantageous.

Environment

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

Requirements

- Conventional approaches decomposed system requirements into subsystem requirements, subsystem requirements into component requirements, and component requirements into unit requirements.
- The equal treatment of all requirements drained away engineering hours from the driving requirements, then wasted that time on paperwork associated with detailed traceability that was inevitably discarded later as the driving requirements and subsequent design understanding evolved.

Requirements

- The ramifications of this approach on the environment's support for requirements management are twofold:

1. The recommended requirements approach is dependent on both textual and model-based representations. Consequently, the environment should provide integrated document automation and visual modeling for capturing textual specifications and use case models.
2. Traceability between requirements and other artifacts needs to be automated. The extent of traceability among sets is the subject of along-standing debate.

Design

- The tools that support the requirements, design, implementation, and assessment workflows are usually used together.
- The primary support required for the design workflow is visual modeling, which is used for capturing design models, presenting them in human-readable format, and translating them into source code.
- An architecture-first and demonstration-based process is enabled by existing architecture components and middleware.

Implementation

- The implementation workflow relies primarily on a programming environment (editor, compiler, debugger, linker, run time) but must also include substantial integration with the change management tools, visual modeling tools, and test automation tools to support productive iteration.

Assessment and Deployment

- The assessment workflow requires all the tools just discussed as well as additional capabilities to support test automation and test management.
- To increase change freedom, testing and document production must be mostly automated.
- **Defect tracking** is another important tool that supports assessment: It provides the change management instrumentation necessary to automate metrics and control release baselines.
- It is also needed to support the deployment workflow throughout the life cycle.

- The Project Environment
  - Round-Trip Engineering
  - Change Management
    - Software Change Orders
    - Configuration Baseline
    - Configuration Control Board
  - Infrastructures
    - Organization Policy
    - Organization Environment
  - Stakeholder Environments

The Project Environment

- The project environment artifacts evolve through three discrete states:
  - The Prototyping Environment
  - The Development Environment
  - The Maintenance Environment

The Project Environment

1. The **Prototyping Environment** includes an architecture test bed for prototyping project architectures to evaluate trade-offs during the inception and elaboration phases of the life cycle.

- This informal configuration of tools should be capable of supporting the following activities:
  - Performance trade-offs and technical risk analyses
  - Make/buy trade-offs and feasibility studies for commercial products
  - Fault tolerance/dynamic reconfiguration trade-offs
  - Analysis of the risks associated with transitioning to full-scale implementation
  - Development of test scenarios, tools, and instrumentation suitable for analyzing the requirements

The Project Environment

2. The **Development Environment** should include a full suite of development tools needed to support the various process workflows and to support round-trip engineering to the maximum extent possible.
3. The **Maintenance Environment** should typically coincide with a mature version of the development environment.
   - In some cases, the **maintenance environment** may be a subset of the **development environment** delivered as one of the project's end products.

The Project Environment

- There are ***four*** important environment disciplines that are critical to the management context and the success of a modern iterative development process:

1. Tools must be integrated to maintain consistency and traceability. ***Round-trip engineering*** is the term used to describe this key requirement for environments that support iterative development.
2. ***Change management*** must be automated and enforced to manage multiple iterations and to enable change freedom. Change is the fundamental primitive of iterative development.

The Project Environment

3. Organizational ***infrastructures*** enable project environments to be derived from a common base of processes and tools. A common infrastructure promotes interproject consistency, reuse of training, reuse of lessons learned, and other strategic improvements to the organization's metaprocess.
4. Extending automation support for ***stakeholder environments*** enables further support for paperless exchange of information and more effective review of engineering artifacts.

Round-Trip Engineering

- As the software industry moves into maintaining different information sets for the engineering artifacts, more automation support is needed to ensure efficient and error-free transition of data from one artifact to another.
- Round-trip engineering is the environment support necessary to maintain consistency among the engineering artifacts.

Round-Trip Engineering

- Below figure depicts some important transitions between information repositories.
- The automated translation of design models to source code (both forward and reverse engineering) is fairly well established.
- The automated translation of design models to process (distribution) models is also becoming straightforward through technologies such as ActiveX and the Common Object Request Broker Architecture(CORBA).
- Compilers and linkers have long provided automation of source code into executable code.

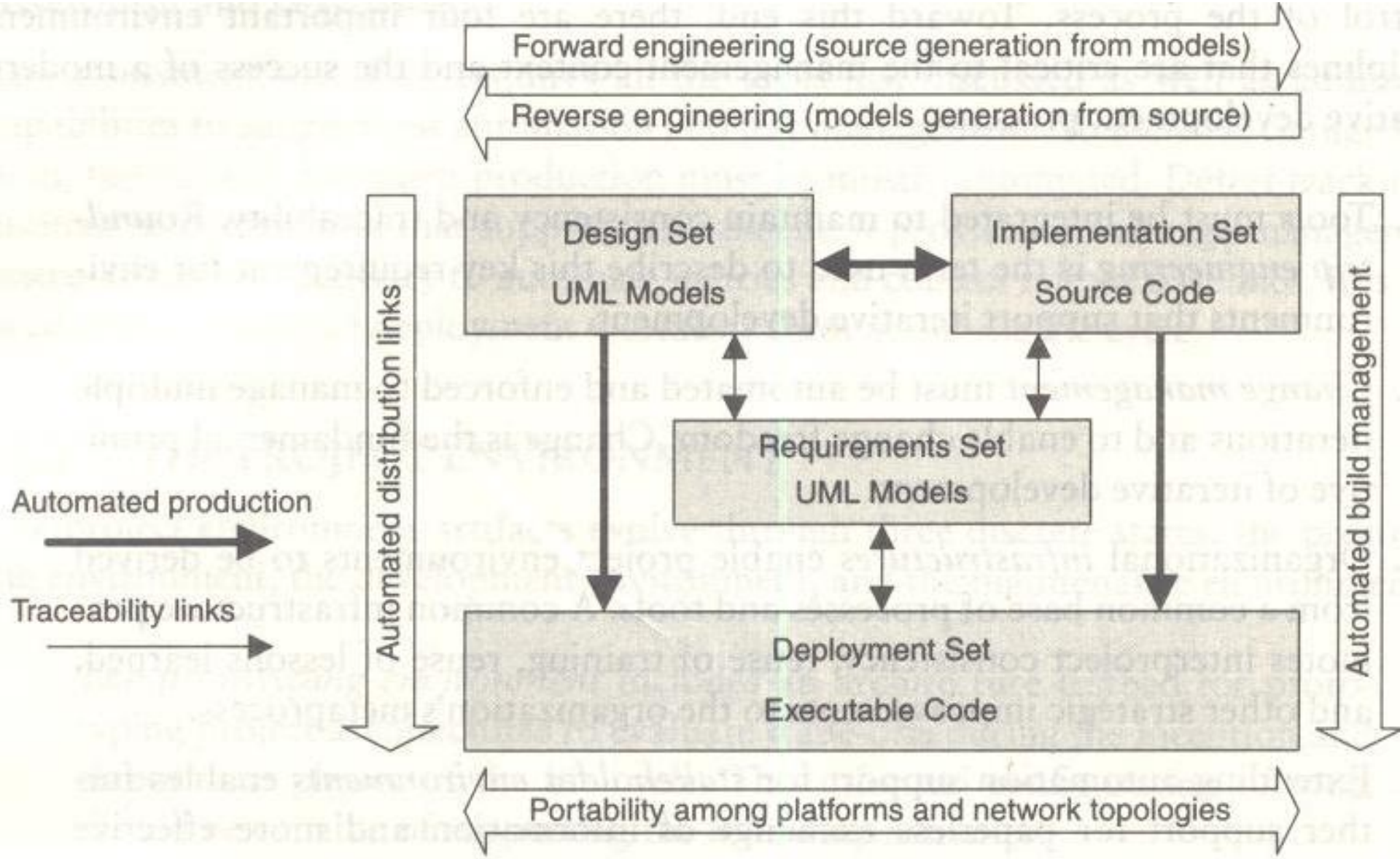


Figure: Round-trip engineering

Round-Trip Engineering

- The primary reason for round-trip engineering is to allow freedom in changing software engineering data sources.
- This configuration control of all the technical artifacts is crucial to maintaining a consistent and error-free representation of the evolving product.
- It is not necessary to have bi-directional transitions in all cases.
- Reverse engineering of poorly constructed legacy source code into an object-oriented design model may be counterproductive.

Round-Trip Engineering

- 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 aspects of the source code representation.
- The code components may still need to be fleshed out with the specifics of certain object attributes or methods.

Change Management

- Change management is as critical to iterative processes as planning.
- Tracking changes in the technical artifacts is crucial to understanding the true technical progress trends and quality trends toward delivering an acceptable end product or interim release.
- In **conventional software management processes**, baseline configuration management techniques for technical artifacts were predominantly a late life-cycle activity.
- In **modern process**—in which requirements, design, and implementation set artifacts are captured in rigorous notations early in the life cycle and are evolved through multiple generations—change management has become fundamental to all phases and almost all activities.

Software Change Orders

- The atomic unit of software work that is authorized to create, modify, or obsolesce components within a configuration baseline is called a **software change order (SCO).**
- Software change orders are a key mechanism for partitioning, allocating, and scheduling software work against an established software baseline and for assessing progress and quality.
- The example SCO shown in below figure is a good starting point for describing a set of change primitives.

Title: _____

Description

Name: _____ Date: _____

Project: _____

Metrics

Category: _____ (0/1 error, 2 enhancement, 3 new feature, 4 other)

Initial Estimate

Breakage: _____

Rework: _____

Actual Rework Expended

Analysis: _____ Test: _____

Implement: _____ Document: _____

Resolution

Analyst: _____

Software Component: _____

Assessment

Method: _____ (inspection, analysis, demonstration, test)

Tester: _____ Platforms: _____ Date: _____

Disposition

State: _____ Release: _____ Priority _____

Acceptance: _____ Date: _____

Closure: _____ Date: _____

Software Change Orders

- The basic fields of the SCO are title, description, metrics, resolution, assessment, and disposition.
  - **Title** - The title is suggested by the originator and is finalized upon acceptance by the configuration control board (CCB). This field should include a reference to an external software problem report if the change was initiated by an external person (such as a user).

Software Change Orders

- **Description** - The problem description includes the name of the originator, date of origination, CCB-assigned SCO identifier, and relevant version identifiers of related support software.
- **Metrics** - The metrics collected for each SCO are important for planning, for scheduling, and for assessing quality improvement. Change categories are type 0 (critical bug), type 1 (bug), type 2 (enhancement), type 3 (new feature), and type 4 (other).

Software Change Orders

- **Resolution** - This field includes the name of the person responsible for implementing the change, the components changed, the actual metrics, and a description of the change.
- **Assessment** - This field describes the assessment technique as either inspection, analysis, demonstration, or test. Where applicable, it should also reference all existing test cases and new test cases executed, and it should identify all different test configurations, such as platforms, topologies, and compilers.

Software Change Orders

- **Disposition** - The SCO is assigned one of the following states by the CCB:
  - **Proposed** : written, pending CCB review
  - **Accepted** : CCB-approved for resolution
  - **Rejected** : closed, with rationale, such as not a problem, duplicate, obsolete change, resolved by another SCO
  - **Archived** : accepted but postponed until a later release
  - **In progress** : assigned and actively being resolved by the development organization
  - **In assessment** : resolved by the development organization; being assessed by a test organization
  - **Closed** : completely resolved, with the concurrence of all CCB members
- A priority and release identifier can also be assigned by the CCB to guide the prioritization and organization of concurrent development activities.

Configuration Baseline

- A configuration baseline is a named collection of software components and supporting documentation that is subject to change management and is upgraded, maintained, tested, statused, and obsolesced as a unit.
- With complex configuration management systems, there are many desirable project-specific and domain-specific standards.

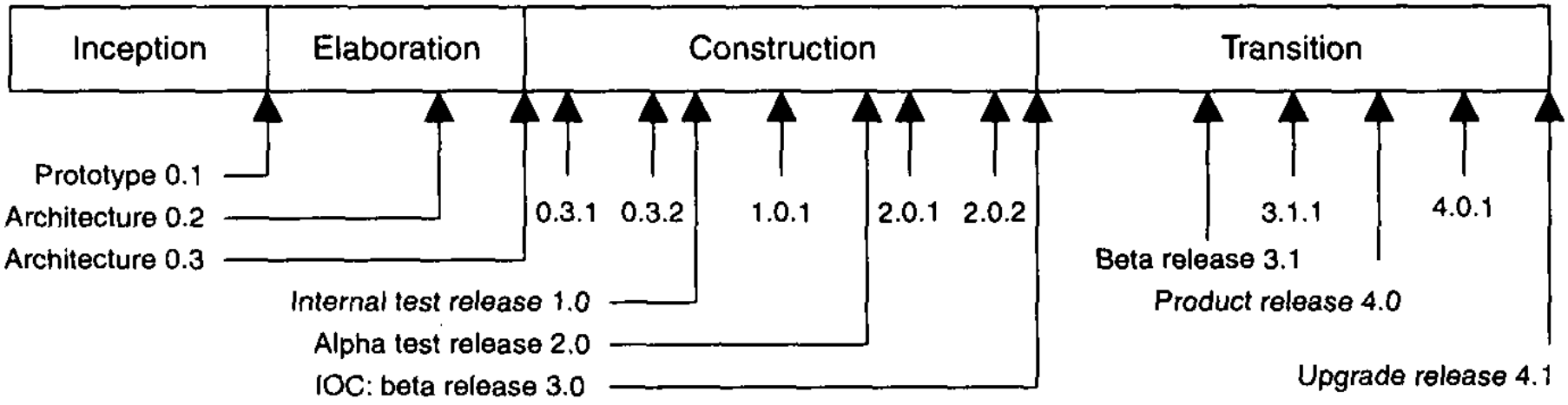
Configuration Baseline

- 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 controlled formally because it is 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 release a configuration baseline to the user community for beta testing.

Configuration Baseline

- 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 new generation of the product or project.
- A **minor release** represents the same basic product but with enhanced features, performance, or quality.
- An **interim release** corresponds to a developmental configuration that is intended to be transient.
- Below figure shows examples of some release name histories for two different situations.

Typical project release sequence for a large-scale, one-of-a-kind project



Typical project release sequence for a small commercial product

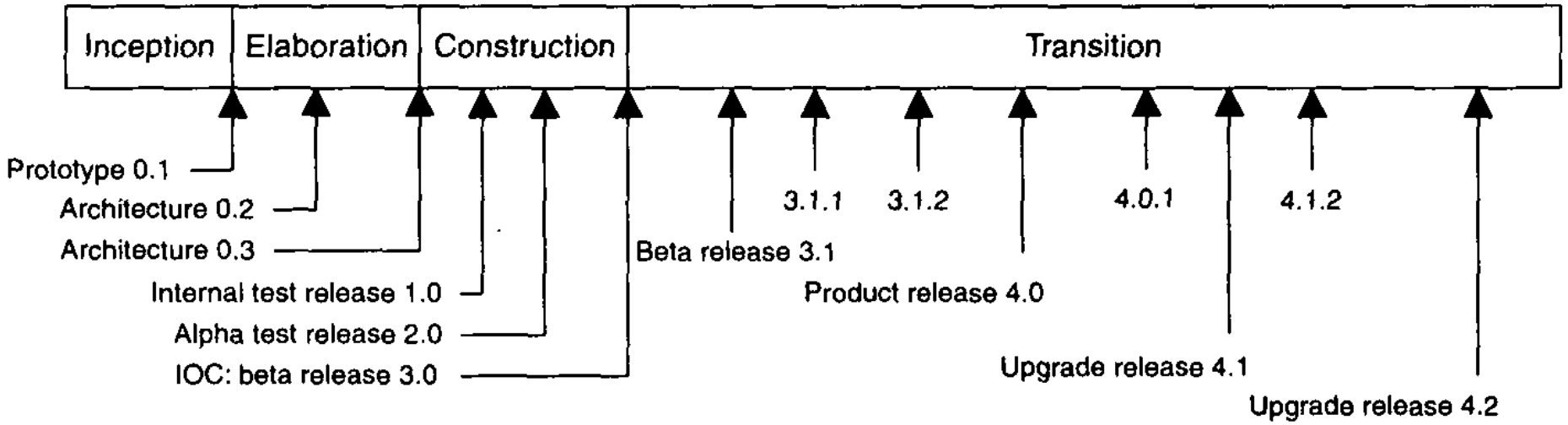


Figure: Example release histories for a typical project and a typical product

Configuration Baseline

- Once software is placed in a controlled baseline, all changes are tracked. A distinction must be made for the cause of a change. Change categories are as follows:
  - **Type 0:** critical failures, which are defects that are nearly always fixed before any external release. In general, these sorts of changes represent showstoppers that have an impact on the usability of the software in its critical use cases.

Configuration Baseline

- **Type 1:** a bug or defect that either does not impair the usefulness of the system or can be worked around. Such errors tend to correlate to nuisances in critical use cases or to serious defects in secondary use cases that have a low probability of occurrence.
- **Type 2:** a change that is an enhancement rather than a response to a defect. Its purpose is typically to improve performance, testability, usability, or some other aspect of quality that represents good value engineering.

Configuration Baseline

- **Type 3:** a change that is necessitated by an update to the requirements. This could be new features or capabilities that are outside the scope of the current vision and business case.
- **Type 4:** changes that are not accommodated by the other categories. Examples include documentation only or a version upgrade to commercial components.

- Below table provides examples of these changes in the context of two different project domains: a large-scale, reliable air traffic control system and a packaged software development tool.

Table: 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 MSExcel 5.0 due to Windows resource management bug

Configuration Control Board

- A CCB is a team of people that functions as the decision authority on the content of configuration baselines.
- A CCB usually includes the software manager, software architecture manager, software development manager, software assessment manager, and other stakeholders (customer, software project manager, systems engineer, user) who are integral to the maintenance of a controlled software delivery system.
- While CCBs typically take action through board meetings, on-line distribution, concurrence, and approval of CCB actions may make sense under many project circumstances.

Configuration Control Board

- The operational concept of an iterative development process must include comprehensive and rigorous change management of the evolving software baselines.
- The fundamental process for controlling the software development and maintenance activities is described through the sequence of states traversed by an SCO.
- The [bracketed] words constitute the state of an SCO transitioning through the process.

Configuration Control Board

- **[Proposed]**. A proposed change is drafted and submitted to the CCB. The proposed change must include a technical description of the problem and an estimate of the resolution effort.
- **[Accepted, archived, or rejected]**. The CCB assigns a unique identifier and accepts, archives, or rejects each proposed change. Acceptance includes the change for resolution in the next release; archiving accepts the change but postpones it for resolution in a future release; and rejection judges the change to be without merit, redundant with other proposed changes, or out of scope.

Configuration Control Board

- **[In progress].** The responsible person analyzes, implements, and tests a solution to satisfy the SCO. This task includes updating documentation, release notes, and SCO metrics actuals, and submitting new SCOs, if necessary. Upon achieving a complete solution, the responsible person completes the resolution section of the SCO and submits it to the independent test team for assessment.
- **[In assessment].** The independent test team assesses whether the SCO is completely resolved. When the independent test team deems the change to be satisfactorily resolved, the SCO is submitted to the CCB for final disposition and closure.
- **[Closed].** When the development organization, independent test organization, and CCB concur that the SCO is resolved, it is transitioned to a closed status.

Infrastructures

- From a process automation perspective, the organization's infrastructure provides the organization's capital assets, including two key artifacts: a **policy** that captures the standards for project software development processes, and an **environment** that captures an inventory of tools.
- These tools are the automation building blocks from which project environments can be configured efficiently and economically.

Organization Policy

- The organization policy is the defining document for the organization's software policies. In any process assessment, this is the tangible(clarity) artifact that says what you do.
- From this document, reviewers should be able to question and review projects and personnel and determine whether the organization does what it says.
- Below figure shows a general outline for an organizational policy.

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: Organization policy outline

Organization Environment

- The **organization environment** for automating the default process will provide many of the answers to how things get done as well as the tools and techniques to automate the process as much as practical.
- Some of the typical components of an organization's automation building blocks are as follows:

Organization Environment

- Standardized tool selections (through investment by the organization in a site license or negotiation of a favorable discount with a tool vendor so that project teams are motivated economically to use that tool), which promote common workflows and a higher ROI on training.
- Standard notations for artifacts, such as UML for all design models, or Ada95 for all custom-developed, reliability-critical implementation artifacts.
- Tool adjuncts such as existing artifact templates (architecture description, evaluation criteria, release descriptions, status assessments) or customizations
- Activity templates (iteration planning, major milestone activities, configuration control boards)

Stakeholder Environments

- The transition to a modern iterative development process with supporting automation should not be restricted to the development team.
- Many large-scale contractual projects include people in external organizations that represent other stakeholders participating in the development process.
- They might include procurement agency contract monitors, end-user engineering support personnel, third-party maintenance contractors, independent verification and validation contractors, representatives of regulatory agencies, and others.

Stakeholder Environments

- These stakeholder representatives also need access to development environment resources so that they can contribute value to the overall effort.
- If an external stake-holder team has no environment resources for accepting on-line products and artifacts, the only vehicle for information exchange is paper.

Stakeholder Environments

- An on-line environment accessible by the external stakeholders allows them to participate in the process as follows:
  - Accept and use executable increments for hands-on evaluation
  - Use the same on-line tools, data, and reports that the software development organization uses to manage and monitor the project
  - Avoid excessive travel, paper interchange delays, format translations, paper and shipping costs, and other overhead costs

Stakeholder Environments

- Below figure illustrates some of the new opportunities for value-added activities by external stakeholders in large contractual efforts.
- There are several important reasons for extending development environment resources into certain stakeholder domains.
  - Technical artifacts are not just paper. Electronic artifacts in rigorous notations such as visual models and source code are viewed far more efficiently by using tools with smart browsers.
  - Independent assessments of the evolving artifacts are encouraged by electronic read-only access to on-line data such as configuration baseline libraries and the change management database. Reviews and inspections, breakage/rework assessments, metrics analyses, and even beta testing can be performed independently of the development team.
  - Even paper documents should be delivered electronically to reduce production costs and turnaround time.

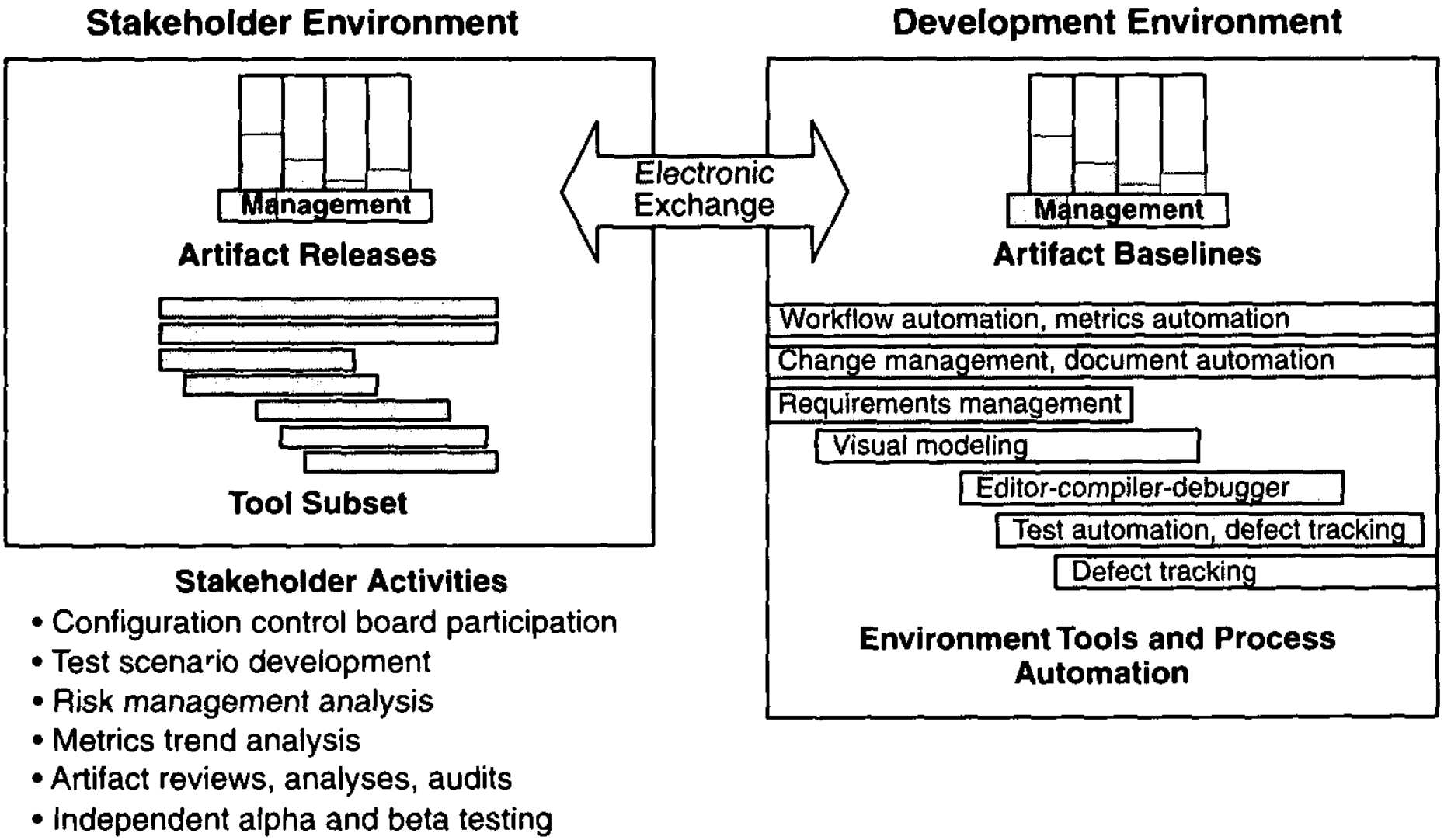


Figure: Extending environments into stakeholder domains

Stakeholder Environments

- Once environment resources are electronically accessible by stakeholders, continuous and expedient feedback is much more efficient, tangible, and useful.
- In implementing such a shared environment, it is important for development teams to create an open environment and provide adequate resources that accommodate customer access.
- It is also important for stakeholders to avoid abusing this access, to participate by adding value, and to avoid interrupting development.
- Internet and intranet technology is making paperless environments economical.