 CHAPTER 6

Artifacts of the Process

Conventional software projects focused on the sequential development of software artifacts: build the requirements, construct a design model traceable to the requirements, build an implementation traceable to the design model, and compile and test the implementation for deployment. This process can work for small-scale, purely custom developments in which the design representation, implementation representation, and deployment representation are closely aligned. For example, a single program that is intended to run on a single computer of a single type and is composed entirely of special-purpose custom components can be constructed with straightforward traceability among all the representations.

However, this approach doesn’t work very well for most of today’s software systems, in which the system complexity (in many dimensions) results in such numerous risks and subtle traceability relationships that you cannot efficiently use a simplistic sequential transformation. Most modern systems are composed of numerous components (some custom, some reused, some commercial products) intended to execute in a heterogeneous network of distributed platforms. They require a very different sequence of artifact evolution and a very different approach to traceability.

Over the past 20 years, the software industry has matured and has transitioned the management process to be iterative. Rather than being built sequentially, the artifacts are evolved together, and the constraints, the different levels of abstractions, and the degrees of freedom are balanced among competing alternatives. Recurring themes

Key Points

▲ The artifacts of the process are organized into five sets: management, requirements, design, implementation, and deployment.
▲ The management artifacts capture the information necessary to synchronize stakeholder expectations.
▲ The requirements, design, implementation, and deployment artifacts are captured in rigorous notations that support automated analysis and browsing.
from successful projects demonstrate that the software artifacts evolve together with balanced levels of detail. Artifacts do not evolve in a one-way, linear progression from requirements to design to implementation to deployment. Choices about implementation and deployment affect the way in which the requirements are stated and the way in which the design proceeds. Information and decisions can flow in various ways among artifacts. The purpose of a good development process is to remove inappropriate, premature constraints on the design and to accommodate the real engineering constraints.

And what is the impact of iterative development on evolving artifacts? The primary difference from the conventional approach is that within each life-cycle phase, the workflow activities do not progress in a simple linear way, nor does artifact building proceed monotonically from one artifact to another. Instead, the focus of activities sweeps across artifacts repeatedly, incrementally enriching the entire system description and the process with the lessons learned in preserving balance across the breadth and depth of information.

### 6.1 THE ARTIFACT SETS

To make the development of a complete software system manageable, distinct collections of information are organized into artifact sets. Each set comprises related artifacts that are persistent and in a uniform representation format (such as English text, C++, Visual Basic, Java, a standard document template, a standard spreadsheet template, or a UML model). While a set represents a complete aspect of the system, an artifact represents cohesive information that typically is developed and reviewed as a single entity. In any given organization, project, or system, some of these artifacts—and even some sets—may be trivial or unnecessary. In general, however, some information needs to be captured in each set to satisfy all stakeholders.

Life-cycle software artifacts are organized into five distinct sets that are roughly partitioned by the underlying language of the set: management (ad hoc textual formats), requirements (organized text and models of the problem space), design (models of the solution space), implementation (human-readable programming language and associated source files), and deployment (machine-processable languages and associated files).

The emergence of rigorous and more powerful engineering notations for requirements and design artifacts that support architecture-first development was a major technology advance. In particular, the Unified Modeling Language has evolved into a suitable representation format, namely visual models with a well-specified syntax and semantics for requirements and design artifacts. Visual modeling using UML is a primitive notation for early life-cycle artifacts. The artifact sets are shown in Figure 6-1; their purposes and notations are described next.
6.1.1 **The Management Set**

The management set captures the artifacts associated with process planning and execution. These artifacts use ad hoc notations, including text, graphics, or whatever representation is required to capture the “contracts” among project personnel (project management, architects, developers, testers, marketers, administrators), among stakeholders (funding authority, user, software project manager, organization manager, regulatory agency), and between project personnel and stakeholders. Specific artifacts included in this set are the work breakdown structure (activity breakdown and financial tracking mechanism), the business case (cost, schedule, profit expectations), the release specifications (scope, plan, objectives for release baselines), the software development plan (project process instance), the release descriptions (results of release baselines), the status assessments (periodic snapshots of project progress), the software change orders (descriptions of discrete baseline changes), the deployment documents (cutover plan, training course, sales rollout kit), and the environment (hardware and software tools, process automation, documentation, training collateral necessary to support the execution of the process described in the software development plan and the production of the engineering artifacts).
ARTIFACTS OF THE PROCESS

Management set artifacts are evaluated, assessed, and measured through a combination of the following:

• Relevant stakeholder review
• Analysis of changes between the current version of the artifact and previous versions (management trends and project performance changes in terms of cost, schedule, and quality)
• Major milestone demonstrations of the balance among all artifacts and, in particular, the accuracy of the business case and vision artifacts

6.1.2 THE ENGINEERING SETS

The engineering sets consist of the requirements set, the design set, the implementation set, and the deployment set. The primary mechanism for evaluating the evolving quality of each artifact set is the transitioning of information from set to set, thereby maintaining a balance of understanding among the requirements, design, implementation, and deployment artifacts. Each of these components of the system description evolves over time.

Requirements Set

Structured text is used for the vision statement, which documents the project scope that supports the contract between the funding authority and the project team. Ad hoc formats may also be used for supplementary specifications (such as regulatory requirements) and user mockups or other prototypes that capture requirements. UML notation is used for engineering representations of requirements models (use case models, domain models). The requirements set is the primary engineering context for evaluating the other three engineering artifact sets and is the basis for test cases.

Requirements artifacts are evaluated, assessed, and measured through a combination of the following:

• Analysis of consistency with the release specifications of the management set
• Analysis of consistency between the vision and the requirements models
• Mapping against the design, implementation, and deployment sets to evaluate the consistency and completeness and the semantic balance between information in the different sets
• Analysis of changes between the current version of requirements artifacts and previous versions (scrap, rework, and defect elimination trends)
• Subjective review of other dimensions of quality
6.1 THE ARTIFACT SETS

**Design Set**

UML notation is used to engineer the design models for the solution. The design set contains varying levels of abstraction that represent the components of the solution space (their identities, attributes, static relationships, dynamic interactions). The design models include enough structural and behavioral information to ascertain a bill of materials (quantity and specification of primitive parts and materials, labor, and other direct costs). Design model information can be straightforwardly and, in many cases, automatically translated into a subset of the implementation and deployment set artifacts. Specific design set artifacts include the design model, the test model, and the software architecture description (an extract of information from the design model that is pertinent to describing an architecture).

The design set is evaluated, assessed, and measured through a combination of the following:

- Analysis of the internal consistency and quality of the design model
- Analysis of consistency with the requirements models
- Translation into implementation and deployment sets and notations (for example, traceability, source code generation, compilation, linking) to evaluate the consistency and completeness and the semantic balance between information in the sets
- Analysis of changes between the current version of the design model and previous versions (scrap, rework, and defect elimination trends)
- Subjective review of other dimensions of quality

Because the level of automated analysis available on design models is currently limited, human analysis must be relied on. This situation should change over the next few years with the maturity of design model analysis tools that support metrics collection, complexity analysis, style analysis, heuristic analysis, and consistency analysis.

**Implementation Set**

The implementation set includes source code (programming language notations) that represents the tangible implementations of components (their form, interface, and dependency relationships) and any executables necessary for stand-alone testing of components. These executables are the primitive parts needed to construct the end product, including custom components, application programming interfaces (APIs) of commercial components, and APIs or reusable or legacy components in a programming language source (such as Ada 95, C++, Visual Basic, Java, or Assembly). Implementation set artifacts can also be translated (compiled and linked) into a subset of
the deployment set (end-target executables). Specific artifacts include self-document-
ing product source code baselines and associated files (compilation scripts, configura-
tion management infrastructure, data files), self-documenting test source code baselines and associated files (input test data files, test result files), stand-alone com-
ponent executables, and component test driver executables.

Implementation sets are human-readable formats that are evaluated, assessed, and measured through a combination of the following:

- Analysis of consistency with the design models
- Translation into deployment set notations (for example, compilation and linking) to evaluate the consistency and completeness among artifact sets
- Assessment of component source or executable files against relevant evalu-
ation criteria through inspection, analysis, demonstration, or testing
- Execution of stand-alone component test cases that automatically compare expected results with actual results
- Analysis of changes between the current version of the implementation set and previous versions (scrap, rework, and defect elimination trends)
- Subjective review of other dimensions of quality

Deployment Set

The deployment set includes user deliverables and machine language notations, ex-
ecutable software, and the build scripts, installation scripts, and executable target-
specific data necessary to use the product in its target environment. These machine
language notations represent the product components in the target form intended for
distribution to users. Deployment set information can be installed, executed against
scenarios of use (tested), and dynamically reconfigured to support the features
required in the end product. Specific artifacts include executable baselines and associ-
ated run-time files, and the user manual.

Deployment sets are evaluated, assessed, and measured through a combination
of the following:

- Testing against the usage scenarios and quality attributes defined in the
requirements set to evaluate the consistency and completeness and the
semantic balance between information in the two sets
- Testing the partitioning, replication, and allocation strategies in mapping
components of the implementation set to physical resources of the deploy-
ment system (platform type, number, network topology)
6.1 The Artifact Sets

- Testing against the defined usage scenarios in the user manual such as installation, user-oriented dynamic reconfiguration, mainstream usage, and anomaly management
- Analysis of changes between the current version of the deployment set and previous versions (defect elimination trends, performance changes)
- Subjective review of other dimensions of quality

The rationale for selecting the management, requirements, design, implementation, and deployment sets was not scientific. The goal was to optimize presentation of the process activities, artifacts, and objectives. Some of the rationale that resulted in this conceptual framework is described next. Although there are several minor exceptions to these generalizations, they are useful in understanding the overall artifact sets.

Each artifact set uses different notation(s) to capture the relevant artifacts. Management set notations (ad hoc text, graphics, use case notation) capture the plans, process, objectives, and acceptance criteria. Requirements notations (structured text and UML models) capture the engineering context and the operational concept. Design notations (in UML) capture the engineering blueprints (architectural design, component design). Implementation notations (software languages) capture the building blocks of the solution in human-readable formats. Deployment notations (executables and data files) capture the solution in machine-readable formats.

Each artifact set is the predominant development focus of one phase of the life cycle; the other sets take on check and balance roles. As illustrated in Figure 6-2, each

![Figure 6-2. Life-cycle focus on artifact sets](image-url)
phase has a predominant focus: Requirements are the focus of the inception phase; design, the elaboration phase; implementation, the construction phase; and deployment, the transition phase. The management artifacts also evolve, but at a fairly constant level across the life cycle.

Most of today’s software development tools map closely to one of the five artifact sets.

1. Management: scheduling, workflow, defect tracking, change management, documentation, spreadsheet, resource management, and presentation tools
2. Requirements: requirements management tools
3. Design: visual modeling tools
4. Implementation: compiler/debugger tools, code analysis tools, test coverage analysis tools, and test management tools
5. Deployment: test coverage and test automation tools, network management tools, commercial components (operating systems, GUIs, DBMSs, networks, middleware), and installation tools

Allocation of responsibilities among project teams is straightforward and aligns with the process workflows presented in Chapter 8.

Implementation Set versus Deployment Set

The separation of the implementation set (source code) from the deployment set (executable code) is important because there are very different concerns with each set. The structure of the information delivered to the user (and typically the test organization) is very different from the structure of the source code information. Engineering decisions that have an impact on the quality of the deployment set but are relatively incomprehensible in the design and implementation sets include the following:

- Dynamically reconfigurable parameters (buffer sizes, color palettes, number of servers, number of simultaneous clients, data files, run-time parameters)
- Effects of compiler/link optimizations (such as space optimization versus speed optimization)
- Performance under certain allocation strategies (centralized versus distributed, primary and shadow threads, dynamic load balancing, hot backup versus checkpoint/rollback)
- Virtual machine constraints (file descriptors, garbage collection, heap size, maximum record size, disk file rotations)
• Process-level concurrency issues (deadlock and race conditions)
• Platform-specific differences in performance or behavior

Much of this configuration information is important engineering source data that should be captured either in the implementation set (if it is embedded within source code) or in the deployment set (if it is embedded within data files, configuration files, installation scripts, or other target-specific components). In dynamically reconfigurable systems or portable components, it is usually better to separate the source code implementation concerns from the target environment concerns (for reasons of performance, dynamic adaptability, or source code change management). With this approach, the implementation can be decoupled from the actual platform type and from the number and topology of the underlying computing infrastructure, which includes operating systems, middleware, networks, and DBMSs.

As an example, consider the software architecture of a one million SLOC missile warning system (a project described in detail in the case study, Appendix D) with extreme requirements for fault tolerance and data processing performance. On this project, significantly different configurations of executables could be built from the same source sets.

• A version that includes only the primary thread of processing on a development host to do a subset of scenario tests
• A version that includes primary and backup processing threads on a development host, which could then exercise some of the logical reconfiguration scenarios
• Functionally equivalent versions of the two preceding configurations that could execute on the target processors to assess the required throughput and response time of the critical-thread scenarios on the candidate target configuration
• A version that could execute a primary thread of servers on one target processor, a shadow thread of servers on a separate backup target processor, a test/exercise thread on either target, and a suite of thread-independent user interface clients on user workstations. The latter, which could support a broad range of dynamic reconfigurations, was essentially the final target configuration.

Deployment of commercial products to customers can also span a broad range of test and deployment configurations. For example, middleware products provide high-performance, reliable object request brokers that are delivered on several platform implementations, including workstation operating systems, bare embedded processors,
large mainframe operating systems, and several real-time operating systems. The product configurations support various compilers and languages as well as various implementations of network software. The heterogeneity of all the various target configurations results in the need for a highly sophisticated source code structure and a huge suite of different deployment artifacts.

6.1.3 Artifact Evolution over the Life Cycle

Each state of development represents a certain amount of precision in the final system description. Early in the life cycle, precision is low and the representation is generally high. Eventually, the precision of representation is high and everything is specified in full detail. At any point in the life cycle, the five sets will be in different states of completeness. However, they should be at compatible levels of detail and reasonably traceable to one another. Performing detailed traceability and consistency analyses early in the life cycle (when precision is low and changes are frequent) usually has a low return on investment. As development proceeds, the architecture stabilizes, and maintaining traceability linkage among artifact sets is worth the effort.

Each phase of development focuses on a particular artifact set. At the end of each phase, the overall system state will have progressed on all sets, as illustrated in Figure 6-3.

The inception phase focuses mainly on critical requirements, usually with a secondary focus on an initial deployment view, little focus on implementation except perhaps choice of language and commercial components, and possibly some high-level focus on the design architecture but not on design detail.

During the elaboration phase, there is much greater depth in requirements, much more breadth in the design set, and further work on implementation and deployment issues such as performance trade-offs under primary scenarios and make/buy analyses. Elaboration phase activities include the generation of an executable

<table>
<thead>
<tr>
<th>Engineering Stage</th>
<th>Production Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inception</td>
<td>Construction</td>
</tr>
<tr>
<td>Elaboration</td>
<td>Transition</td>
</tr>
</tbody>
</table>

**Figure 6-3.** Life-cycle evolution of the artifact sets
prototype. This prototype involves subsets of development in all four sets and specifically assesses whether the interfaces and collaborations among components are consistent and complete within the context of the system’s primary requirements and scenarios. Although there is generally a broad understanding of component interfaces, there is usually not much depth in implementation for custom components. (However, commercial or other existing components may be fully elaborated.) A portion of all four sets must be evolved to some level of completion before an architecture baseline can be established. This evolution requires sufficient assessment of the design set, implementation set, and deployment set artifacts against the critical use cases of the requirements set to suggest that the project can proceed predictably with well-understood risks.

The main focus of the construction phase is design and implementation. The main focus early in this phase should be the depth of the design artifacts. Later in construction, the emphasis is on realizing the design in source code and individually tested components. This phase should drive the requirements, design, and implementation sets almost to completion. Substantial work is also done on the deployment set, at least to test one or a few instances of the programmed system through a mechanism such as an alpha or beta release.

The main focus of the transition phase is on achieving consistency and completeness of the deployment set in the context of the other sets. Residual defects are resolved, and feedback from alpha, beta, and system testing is incorporated.

As development proceeds, each of the parts evolves in more detail. When the system is complete, all four sets are fully elaborated and consistent with one another. In contrast to the conventional practice, you do not specify the requirements, then do the design, and so forth. Instead, you evolve the entire system; decisions about the deployment may affect requirements, not just the other way around. The key emphasis here is to break the conventional mold, in which the default interpretation is that one set precedes another. Instead, one state of the entire system evolves into a more elaborate state of the system, usually involving evolution in each of the parts. During the transition phase, traceability between the requirements set and the deployment set is extremely important. The evolving requirements set captures a mature and precise representation of the stakeholders’ acceptance criteria, and the deployment set represents the actual end-user product. Therefore, during the transition phase, completeness and consistency between these two sets are important. Traceability among the other sets is necessary only to the extent that it aids the engineering or management activities.

6.1.4 Test Artifacts

Conventional software testing followed the same document-driven approach that was applied to software development. Development teams built requirements documents,
top-level design documents, and detailed design documents before constructing any source files or executables. Similarly, test teams built system test plan documents, system test procedure documents, integration test plan documents, unit test plan documents, and unit test procedure documents before building any test drivers, stubs, or instrumentation. This document-driven approach caused the same problems for the test activities that it did for the development activities.

One of the truly discriminating tenets of a modern process is to use exactly the same sets, notations, and artifacts for the products of test activities as are used for product development. In essence, we are simply identifying the test infrastructure necessary to execute the test process as a required subset of the end product. By doing this, we have forced several engineering disciplines into the process.

- The test artifacts must be developed concurrently with the product from inception through deployment. Thus, testing is a full-life-cycle activity, not a late life-cycle activity.
- The test artifacts are communicated, engineered, and developed within the same artifact sets as the developed product.
- The test artifacts are implemented in programmable and repeatable formats (as software programs).
- The test artifacts are documented in the same way that the product is documented.
- Developers of the test artifacts use the same tools, techniques, and training as the software engineers developing the product.

These disciplines allow for significant levels of homogenization across project workflows, which are described in Chapter 8. Everyone works within the notations and techniques of the four sets used for engineering artifacts, rather than with separate sequences of design and test documents. Interpersonal communications, stakeholder reviews, and engineering analyses can be performed with fewer distinct formats, fewer ad hoc notations, less ambiguity, and higher efficiency.

Testing is only one aspect of the assessment workflow. Other aspects include inspection, analysis, and demonstration. Testing refers to the explicit evaluation through execution of deployment set components under a controlled scenario with an expected and objective outcome. The success of a test can be determined by comparing the expected outcome to the actual outcome with well-defined mathematical precision. Tests are exactly the forms of assessment that are automated.

Although the test artifact subsets are highly project-specific, the following example clarifies the relationship between test artifacts and the other artifact sets. Consider a project to perform seismic data processing for the purpose of oil exploration. This
system has three fundamental subsystems: (1) a sensor subsystem that captures raw seismic data in real time and delivers these data to (2) a technical operations subsystem that converts raw data into an organized database and manages queries to this database from (3) a display subsystem that allows workstation operators to examine seismic data in human-readable form. Such a system would result in the following test artifacts:

- **Management set.** The release specifications and release descriptions capture the objectives, evaluation criteria, and results of an intermediate milestone. These artifacts are the test plans and test results negotiated among internal project teams. The software change orders capture test results (defects, testability changes, requirements ambiguities, enhancements) and the closure criteria associated with making a discrete change to a baseline.

- **Requirements set.** The system-level use cases capture the operational concept for the system and the acceptance test case descriptions, including the expected behavior of the system and its quality attributes. The entire requirements set is a test artifact because it is the basis of all assessment activities across the life cycle.

- **Design set.** A test model for nondeliverable components needed to test the product baselines is captured in the design set. These components include such design set artifacts as a seismic event simulation for creating realistic sensor data; a “virtual operator” that can support unattended, after-hours test cases; specific instrumentation suites for early demonstration of resource usage; transaction rates or response times; and use case test drivers and component stand-alone test drivers.

- **Implementation set.** Self-documenting source code representations for test components and test drivers provide the equivalent of test procedures and test scripts. These source files may also include human-readable data files representing certain statically defined data sets that are explicit test source files. Output files from test drivers provide the equivalent of test reports.

- **Deployment set.** Executable versions of test components, test drivers, and data files are provided.

For any release, all the test artifacts and product artifacts are maintained using the same baseline version identifier. They are created, changed, and obsolesced as a consistent unit. Because test artifacts are captured using the same notations, methods, and tools, the approach to testing is consistent with design and development. This approach forces the evolving test artifacts to be maintained so that regression testing can be automated easily.
6.2 MANAGEMENT ARTIFACTS

The management set includes several artifacts that capture intermediate results and ancillary information necessary to document the product/process legacy, maintain the product, improve the product, and improve the process. These artifacts are summarized next and discussed in detail in subsequent chapters, where the management workflows and activities are elaborated. Although the following descriptions use the word document to describe certain artifacts, this is only meant to imply that the data could be committed to a paper document. In many cases, the data may be processed, reviewed, and exchanged via electronic means only.

Business Case

The business case artifact provides all the information necessary to determine whether the project is worth investing in. It details the expected revenue, expected cost, technical and management plans, and backup data necessary to demonstrate the risks and realism of the plans. In large contractual procurements, the business case may be implemented in a full-scale proposal with multiple volumes of information. In a small-scale endeavor for a commercial product, it may be implemented in a brief plan with an attached spreadsheet. The main purpose is to transform the vision into economic terms so that an organization can make an accurate ROI assessment. The financial forecasts are evolutionary, updated with more accurate forecasts as the life cycle progresses. Figure 6-4 provides a default outline for a business case.

Software Development Plan

The software development plan (SDP) elaborates the process framework into a fully detailed plan. It is the defining document for the project’s process. It must comply with the contract (if any), comply with organization standards (if any), evolve along

| I.  | Context (domain, market, scope) |
| II. | Technical approach            |
|     | A. Feature set achievement plan |
|     | B. Quality achievement plan    |
|     | C. Engineering trade-offs and technical risks |
| III. | Management approach         |
|     | A. Schedule and schedule risk assessment |
|     | B. Objective measures of success |
| IV.  | Evolutionary appendixes       |
|     | A. Financial forecast         |
|     | 1. Cost estimate             |
|     | 2. Revenue estimate          |
|     | 3. Bases of estimates        |

**Figure 6-4.** Typical business case outline
with the design and requirements, and be used consistently across all subordinate organizations doing software development. Two indications of a useful SDP are periodic updating (it is not stagnant shelfware) and understanding and acceptance by managers and practitioners alike. Figure 6-5 provides a default outline for a software development plan.

**Work Breakdown Structure**

A work breakdown structure (WBS) is the vehicle for budgeting and collecting costs. To monitor and control a project’s financial performance, the software project manager must have insight into project costs and how they are expended. The structure of cost accountability is a serious project planning constraint. Lessons learned in numerous less-than-successful projects have shown that if the WBS is structured improperly, it can drive the evolving design and product structure in the wrong direction. A project manager should not lay out lower levels of a WBS (thereby defining specific boundaries of accountability) until a commensurate level of stability in the product structure is achieved. A functional breakdown in the WBS will result in a functional decomposition in the software. The concept of an evolutionary WBS is developed further in Chapter 10.

**Figure 6-5.** Typical software development plan outline
Software Change Order Database

Managing change is one of the fundamental primitives of an iterative development process. With greater change freedom, a project can iterate more productively. This flexibility increases the content, quality, and number of iterations that a project can achieve within a given schedule. Change freedom has been achieved in practice through automation, and today's iterative development environments carry the burden of change management. Organizational processes that depend on manual change management techniques have encountered major inefficiencies. Consequently, the change management data have been elevated to a first-class management artifact that is described as a database to instill the concept of a need for automation. Once software is placed in a controlled baseline, all changes must be formally tracked and managed. By automating data entry and maintaining change records on-line, most of the change management bureaucracy and metrics collection and reporting activities can be automated. Software change orders are discussed in detail in Chapter 12.

Release Specifications

The scope, plan, and objective evaluation criteria for each baseline release are derived from the vision statement as well as many other sources (make/buy analyses, risk management concerns, architectural considerations, shots in the dark, implementation constraints, quality thresholds). These artifacts are intended to evolve along with the process, achieving greater fidelity as the life cycle progresses and requirements understanding matures. Figure 6-6 provides a default outline for a release specification.

There are two important forms of requirements. The first is the vision statement (or user need), which captures the contract between the development group and the buyer. This information should be evolving, but varying slowly, across the life cycle. It should be represented in a form that is understandable to the buyer (an ad hoc format

<table>
<thead>
<tr>
<th>I.</th>
<th>Iteration content</th>
</tr>
</thead>
<tbody>
<tr>
<td>II.</td>
<td>Measurable objectives</td>
</tr>
<tr>
<td></td>
<td>A. Evaluation criteria</td>
</tr>
<tr>
<td></td>
<td>B. Followthrough approach</td>
</tr>
<tr>
<td>III.</td>
<td>Demonstration plan</td>
</tr>
<tr>
<td></td>
<td>A. Schedule of activities</td>
</tr>
<tr>
<td></td>
<td>B. Team responsibilities</td>
</tr>
<tr>
<td>IV.</td>
<td>Operational scenarios (use cases demonstrated)</td>
</tr>
<tr>
<td></td>
<td>A. Demonstration procedures</td>
</tr>
<tr>
<td></td>
<td>B. Traceability to vision and business case</td>
</tr>
</tbody>
</table>

Figure 6-6. Typical release specification outline
that may include text, mockups, use cases, spreadsheets, or other formats). A use case model in the vision statement context serves to capture the operational concept in terms the user/buyer will understand.

Evaluation criteria, the second form of requirements contained in release specifications, are transient snapshots of objectives for a given intermediate life-cycle milestone. Evaluation criteria in release specifications are defined as management artifacts rather than as part of the requirements set. They are derived from the vision statement as well as many other sources (make/buy analyses, risk management concerns, architectural considerations, shots in the dark, implementation constraints, quality thresholds). These management-oriented requirements may be represented by use cases, use case realizations, annotations on use cases, or structured text representations.

The system requirements (user/buyer concerns) are captured in the vision statement. Lower levels of requirements are driven by the process (organized by iteration rather than by lower level component) in the form of evaluation criteria (typically captured by a set of use cases and other textually represented objectives). Thus, the lower level requirements can evolve as summarized in the following conceptual example for a relatively large project:

1. Inception iterations. Typically, 10 to 20 evaluation criteria capture the driving issues associated with the critical use cases that have an impact on architecture alternatives and the overall business case.

2. Elaboration iterations. These evaluation criteria (perhaps 50 or so), when demonstrated against the candidate architecture, verify that the critical use cases and critical requirements of the vision statement can be met with low risk.

3. Construction iterations. These evaluation criteria (perhaps hundreds) associated with a meaningful set of use cases, when passed, constitute useful subsets of the product that can be transitioned to formal test or to alpha or beta releases.

4. Transition iterations. This complete set of use cases and associated evaluation criteria (perhaps thousands) constitutes the acceptance test criteria associated with deploying a version into operation.

This process is naturally evolutionary and is loosely coupled to the actual design and architecture that evolves. In the end, 100% traceability becomes important, but intermediate activities and milestones are far less concerned with consistency and completeness than they were when the conventional approach to software development was used. Each iteration’s evaluation criteria are discarded once the milestone is completed; they are transient artifacts. A better version is created at each stage, so
there is much conservation of content and lessons learned in each successive set of evaluation criteria. Release specification artifacts and their inherent evaluation criteria are more concerned early on with ensuring that the highest risk issues are resolved.

**Release Descriptions**

Release description documents describe the results of each release, including performance against each of the evaluation criteria in the corresponding release specification. Release baselines should be accompanied by a release description document that describes the evaluation criteria for that configuration baseline and provides substantiation (through demonstration, testing, inspection, or analysis) that each criterion has been addressed in an acceptable manner. This document should also include a metrics summary that quantifies its quality in absolute and relative terms (compared to the previous versions, if any). The results of a post-mortem review of any release would be documented here, including outstanding issues, recommendations for process and product improvement, trade-offs in addressing evaluation criteria, follow-up actions, and similar information. Figure 6-7 provides a default outline for a release description.

**Status Assessments**

Status assessments provide periodic snapshots of project health and status, including the software project manager’s risk assessment, quality indicators, and management indicators. Although the period may vary, the forcing function needs to persist. The paramount objective of a good management process is to ensure that the expectations of all stakeholders (contractor, customer, user, subcontractor) are synchronized and consistent. The periodic status assessment documents provide the critical mechanism for managing everyone’s expectations throughout the life cycle; for addressing, communicating, and resolving management issues, technical issues, and project risks; and

![Figure 6-7. Typical release description outline](image-url)
for capturing project history. They are the periodic heartbeat for management attention. Section 9.3 discusses status assessments in more detail.

Typical status assessments should include a review of resources, personnel staffing, financial data (cost and revenue), top 10 risks, technical progress (metrics snapshots), major milestone plans and results, total project or product scope, action items, and follow-through. Continuous open communications with objective data derived directly from on-going activities and evolving product configurations are mandatory in any project.

**Environment**

An important emphasis of a modern approach is to define the development and maintenance environment as a first-class artifact of the process. A robust, integrated development environment must support automation of the development process. This environment should include requirements management, visual modeling, document automation, host and target programming tools, automated regression testing, and continuous and integrated change management, and feature and defect tracking. A common theme from successful software projects is that they hire good people and provide them with good tools to accomplish their jobs. Automation of the software development process provides payback in quality, the ability to estimate costs and schedules, and overall productivity using a smaller team. By allowing the designers to traverse quickly among development artifacts and easily keep the artifacts up-to-date, integrated toolsets play an increasingly important role in incremental and iterative development.

**Deployment**

A deployment document can take many forms. Depending on the project, it could include several document subsets for transitioning the product into operational status. In big contractual efforts in which the system is delivered to a separate maintenance organization, deployment artifacts may include computer system operations manuals, software installation manuals, plans and procedures for cutover (from a legacy system), site surveys, and so forth. For commercial software products, deployment artifacts may include marketing plans, sales rollout kits, and training courses.

**Management Artifact Sequences**

In each phase of the life cycle, new artifacts are produced and previously developed artifacts are updated to incorporate lessons learned and to capture further depth and breadth of the solution. Some artifacts are updated at each major milestone, others at each minor milestone. Figure 6-8 identifies a typical sequence of artifacts across the life-cycle phases.


ARTIFACTS OF THE PROCESS

![Informal version](image1)
![Controlled baseline](image2)

<table>
<thead>
<tr>
<th>Artifact Set</th>
<th>Inception</th>
<th>Elaboration</th>
<th>Construction</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Work breakdown structure</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
</tr>
<tr>
<td>Business case</td>
<td>▲</td>
<td>▲</td>
<td></td>
<td>▲</td>
</tr>
<tr>
<td>Release specifications</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td>Software development plan</td>
<td>▲</td>
<td>▲</td>
<td></td>
<td>▲</td>
</tr>
<tr>
<td>Release descriptions</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td>Status assessments</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td>Software change order data</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
</tr>
<tr>
<td>Deployment documents</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>▲</td>
</tr>
<tr>
<td>Environment</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td>Vision document</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
</tr>
<tr>
<td>Requirements model(s)</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
</tr>
<tr>
<td>Design model(s)</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>▲</td>
</tr>
<tr>
<td>Test model</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>▲</td>
</tr>
<tr>
<td>Architecture description</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>▲</td>
</tr>
<tr>
<td>Source code baselines</td>
<td>▲</td>
<td>▲</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td>Associated compile-time files</td>
<td>▲</td>
<td>▲</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td>Component executables</td>
<td>▲</td>
<td>▲</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td>Integrated product-executable baselines</td>
<td>▲</td>
<td>▲</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td>Associated run-time files</td>
<td>▲</td>
<td>▲</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td>User manual</td>
<td>△</td>
<td>△</td>
<td>△</td>
<td>▲</td>
</tr>
</tbody>
</table>

**Figure 6-8.** Artifact sequences across a typical life cycle
6.3 ENGINEERING ARTIFACTS

Most of the engineering artifacts are captured in rigorous engineering notations such as UML, programming languages, or executable machine codes. Because this book is written from a management perspective, it does not dwell on these artifacts. However, three engineering artifacts are explicitly intended for more general review, and they deserve further elaboration.

Vision Document

The vision document provides a complete vision for the software system under development and supports the contract between the funding authority and the development organization. Whether the project is a huge military-standard development (whose vision could be a 300-page system specification) or a small, internally funded commercial product (whose vision might be a two-page white paper), every project needs a source for capturing the expectations among stakeholders. A project vision is meant to be changeable as understanding evolves of the requirements, architecture, plans, and technology. A good vision document should change slowly. Figure 6-9 provides a default outline for a vision document.

The vision document is written from the user’s perspective, focusing on the essential features of the system and acceptable levels of quality. The vision document should contain at least two appendixes. The first appendix should describe the operational concept using use cases (a visual model and a separate artifact). The second appendix should describe the change risks inherent in the vision statement, to guide defensive design efforts.

The vision statement should include a description of what will be included as well as those features considered but not included. It should also specify operational capacities (volumes, response times, accuracies), user profiles, and interoperation interfaces with entities outside the system boundary, where applicable. The vision should not be defined only for the initial operating level; its likely evolution path

<table>
<thead>
<tr>
<th>I. Feature set description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A. Precedence and priority</td>
</tr>
<tr>
<td>II. Quality attributes and ranges</td>
</tr>
<tr>
<td>III. Required constraints</td>
</tr>
<tr>
<td>A. External interfaces</td>
</tr>
<tr>
<td>IV. Evolutionary appendices</td>
</tr>
<tr>
<td>A. Use cases</td>
</tr>
<tr>
<td>1. Primary scenarios</td>
</tr>
<tr>
<td>2. Acceptance criteria and tolerances</td>
</tr>
<tr>
<td>B. Desired freedoms (potential change scenarios)</td>
</tr>
</tbody>
</table>

Figure 6-9. Typical vision document outline
should be addressed so that there is a context for assessing design adaptability. The operational concept involves specifying the use cases and scenarios for nominal and off-nominal usage. The use case representation provides a dynamic context for understanding and refining the scope, for assessing the integrity of a design model, and for developing acceptance test procedures. The vision document provides the contractual basis for the requirements visible to the stakeholders.

**Architecture Description**

The architecture description provides an organized view of the software architecture under development. It is extracted largely from the design model and includes views of the design, implementation, and deployment sets sufficient to understand how the operational concept of the requirements set will be achieved. The breadth of the architecture description will vary from project to project depending on many factors. The architecture can be described using a subset of the design model or as an abstraction of the design model with supplementary material, or a combination of both. As examples of these two forms of descriptions, consider the architecture of this book:

- A subset form could be satisfied by the table of contents. This description of the architecture of the book is directly derivable from the book itself.
- An abstraction form could be satisfied by a "Cliffs Notes" treatment. (Cliffs Notes are condensed versions of classic books used as study guides by some college students.) This format is an abstraction that is developed separately and includes supplementary material that is not directly derivable from the evolving product.

The approach described in Section 7.2 allows an architecture description to be tailored to the specific needs of a project. Figure 6-10 provides a default outline for an architecture description.

**Software User Manual**

The software user manual provides the user with the reference documentation necessary to support the delivered software. Although content is highly variable across application domains, the user manual should include installation procedures, usage procedures and guidance, operational constraints, and a user interface description, at a minimum. For software products with a user interface, this manual should be developed early in the life cycle because it is a necessary mechanism for communicating and stabilizing an important subset of requirements. The user manual should be written by members of the test team, who are more likely to understand the user's perspective than the development team. If the test team is responsible for the manual, it can be generated in parallel with development and can be evolved early as a tangible and rel-
Conventional document-driven approaches squandered incredible amounts of engineering time on developing, polishing, formatting, reviewing, updating, and distributing documents. Why? There are several reasons that documents became so important to the process. First, there were no rigorous engineering methods or languages for requirements specification or design. Consequently, paper documents with ad hoc text and graphical representations were the default format. Second, conventional languages of implementation and deployment were extremely cryptic and highly unstructured. To present the details of software structure and behavior to other interested reviewers (testers, maintainers, managers), a more human-readable format was needed. Probably most important, software progress needed to be "credibly" assessed. Documents represented a tangible but misleading mechanism for demonstrating progress.

In some domains, document-driven approaches have degenerated over the past 30 years into major obstacles to process improvement. The quality of the documents became more important than the quality of the engineering information they represented. And evaluating quality through human review of abstract descriptions is a highly subjective process. Much effort was expended assessing single-dimensional surface issues, with very little attention devoted to the multidimensional issues that drive architecture qualities, such as performance and adaptability.

**Figure 6-10.** Typical architecture description outline
Document production cycles, review cycles, and update cycles also injected very visible and formal snapshots of progress into the schedule, thereby introducing more schedule dependencies and synchronization points. For example, the following scenario was not uncommon on large defense projects: Spend a month preparing a design document, deliver the document to the customer for review, wait a month to receive comments back, then spend a month responding to comments and incorporating changes. With many, many multiple-month document review cycles to be managed, scheduled, and synchronized, it is not surprising that many such projects ended up with five-year development life cycles. Lengthy and highly detailed documents, which were generally perceived to demonstrate more progress, resulted in premature engineering details and increased scrap and rework later in the life cycle.

A more effective approach is to redirect this documentation effort to improving the rigor and understandability of the information source and allowing on-line review of the native information source by using smart browsing and navigation tools. Such an approach can eliminate a huge, unproductive source of scrap and rework in the process and allow for continuous review by everyone who is directly concerned with the evolving on-line artifacts.

This philosophy raises the following cultural issues:

- **People want to review information but don't understand the language of the artifact.** Many interested reviewers of a particular artifact will resist having to learn the engineering language in which the artifact is written. It is not uncommon to find people (such as veteran software managers, veteran quality assurance specialists, or an auditing authority from a regulatory agency) who react as follows: “I’m not going to learn UML, but I want to review the design of this software, so give me a separate description such as some flowcharts and text that I can understand.” Would we respond to a similar request by someone reviewing the engineering blueprints of a building? No. We would require that the reviewer be knowledgeable in the engineering notation. We should stop patronizing audiences who resist treating software as an engineering discipline. These interested parties typically add cost and time to the process without adding value.

- **People want to review the information but don't have access to the tools.** It is not very common for the development organization to be fully tooled; it is extremely rare that the other stakeholders have any capability to review the engineering artifacts on-line. Consequently, organizations are forced to exchange paper documents. Standardized formats (such as UML, spreadsheets, Visual Basic, C++, and Ada 95), visualization tools, and the Web are rapidly making it economically feasible for all stakeholders to exchange information electronically. The approach to artifacts is one area in which
the optimal software development process can be polluted if the philosophy of the process is not accepted by the other stakeholders.

- **Human-readable engineering artifacts should use rigorous notations that are complete, consistent, and used in a self-documenting manner.** Properly spelled English words should be used for all identifiers and descriptions. Acronyms and abbreviations should be used only where they are well-accepted jargon in the context of the component's usage. No matter what languages or tools are used, there is no reason to abbreviate and encrypt modeling or programming language source identifiers. Saving keystrokes through abbreviation may simplify the artifact author's job, but it introduces errors throughout the rest of the life cycle. Disallowing this practice will pay off in both productivity and quality. Software is written only once, but it is read many times. Therefore, readability should be emphasized and the use of proper English words should be required in all engineering artifacts. This practice enables understandable representations, browseable formats (paperless review), more-rigorous notations, and reduced error rates.

- **Useful documentation is self-defining: It is documentation that gets used.** Above all, building self-documenting engineering artifacts gives the development organization the "right" to work solely in the engineering notations and avoid separate documents to describe all the details of a model, component, or test procedure. If you find that information, and particularly a document, is getting produced but not used, eliminate it in favor of whatever is getting used to accomplish the intended purpose. Strive to improve its self-documenting nature.

- **Paper is tangible; electronic artifacts are too easy to change.** One reason some stakeholders prefer paper documents is that once they are delivered, they are tangible, static, and persistent. On-line and Web-based artifacts can be changed easily and are viewed with more skepticism because of their inherent volatility. Although electronic artifacts will and should be met with healthy skepticism by many stakeholders, it is simply a matter of time before the whole world operates this way. The advantages are substantial and far-reaching across many domains. Rest assured that tools and environments will evolve to support change management, audit trails, electronic signatures, and other advances in groupware so that electronic interchange replaces paper.

   It is extremely important that the information inherent in the artifact be emphasized, not the paper on which it is written. Short documents are usually more useful than long ones. Software is the primary product; documentation is merely support material.