★Concept: Architecture

- ☑#concept# Software Architecture
- ☑#concept# Architectural Constraints
- ☑#concept# Architectural Goals
- ☑#concept# Architecturally Significant Requirements
- ☑#concept# Architectural Views and Viewpoints
- ☑#concept# Executable Architecture
- ☑#concept# Key Abstractions
- ☑#concept# Architectural Mechanism
- ☑#concept# Analysis Mechanism
- ☑#concept# Design Mechanism
- ☑#concept# Implementation Mechanism
-

.

.

#concept# Software Architecture

Summary
The software architecture represents the structure or structures of the system, which consists of software **components**, the externally visible **properties** of those components, and the **relationships** among them.

Main Description

- **Introduction**

  Software architecture is a concept that is easy to understand, and that most engineers intuitively feel, especially with a little experience, but it is hard to define precisely. In particular, it is difficult to draw a sharp line between design and architecture-architecture is one aspect of design that concentrates on some specific features.

  In An Introduction to Software Architecture, David Garlan and Mary Shaw suggest that software architecture is a level of design concerned with issues: "Beyond the

algorithms and **data structures** of the computation; designing and specifying the overall system structure emerges as a new kind of problem. **Structural issues** include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives." [GAR93]

But there is more to architecture than just structure; the IEEE Working Group on Architecture defines it as "**the highest-level concept of a system in its environment**" [IEP1471]. It also encompasses the "fit" with system integrity, with economical constraints, with aesthetic concerns, and with style. It is not limited to an inward focus, but takes into consideration the system as a whole in its user environment and its development environment - an outward focus.

The architecture focuses on specific aspects of the overall system design, concentrating on **structure, essential elements, key scenarios** and those aspects that have a lasting impact on **system qualities such as performance, reliability, adaptability and cost.** It also defines the set of **Architectural Mechanisms, Patterns and styles** that will guide the rest of the design, assuring its integrity.

- **Purpose of Architecture**

  The architecture can be used for many things:

  - To describe the essential structure of the system and the decisions guiding the structure of the system so the integrity and understandability of the system is assured.
  - To identify and attack risks to the system (using the architecture as an artifact of governance)
  - To provide context and guidance for developers to construct the system by describing the motivations behind the architectural decisions so those decisions can be robustly implemented. The architecture services

as the blueprint for development. For example, the architect may place constraints on how data is packaged and communicated between different parts of the system. This may appear to be a burden, but the justification in the Architecture Notebook can explain that there is a significant performance bottleneck when communicating with a legacy system. The rest of the system must adapt to this bottleneck by following a specific data packaging scheme.
    - To provide an overview of the system to whoever must maintain the architecture, as well as an understanding of the motivation behind the important technical decisions. Team members who were not involved in those architectural decisions need to understand the reasoning behind the context of the architecture so they can best address the needs of the system.
    - To define the project structure and team organization. Architectural elements make excellent units of implementation, unit testing, integration, configuration management and documentation. They can also be used to define so that managers can plan the project.

- **Architecture Description**

  To speak and reason about software architecture, you must first define an architectural representation, a way of describing important aspects of an architecture.

  The following is some information that is worth capturing as part of the software architecture:

    - Architectural goals (see **#concept# Architectural Goals**)
    - References to architecturally significant requirements and how the architecture addresses those requirements, including key scenarios that describe critical behavior of the system (see **#concept# Architecturally Significant Requirements**)
    - Constraints on the architecture and how the architecture addresses those constraints (see **#concept# Architectural Constraints**)
    - Key abstractions (see **#concept# Key Abstractions**)

- Architectural Mechanism and where they should be applied (see **#concept# Architectural Mechanism**).
- Description of the **partitioning** approach, as well as a description of the key partitions. For example, Layering (see **#guideline# Layering**)
- Description of the **deployment** approach, as well as how key components are allocated to deployment nodes.
- References to architecturally significant design elements (see **#concept# Component**)
- Critical system interfaces (see **#guideline# Representing Interfaces to External Systems**)
- Assets that have been reused and/or assets that have been developed to be reused (for more information, see **#guideline# Software Reuse**)
- Guidance, decisions, and constraints the developers must follow in building the system, along with justification.

The architecture can contain any information and references that are appropriate in communicating how developers should build the system.

- **Architectural Representation**

  The architecture can be represented in many forms and from many viewpoints, depending on the needs of the project and the preferences of the project team. It need not be a formal document. The essence of the architecture can often be communicated through a series of simple diagrams on a whiteboard; or as a list of decisions. The illustration just needs to show the nature of the proposed solution, convey the **governing ideas**, and represent the **major building blocks** to make it easier to communicate the architecture to the project team and stakeholders.

  If a more complex system is required, then the architecture can be represented as a more comprehensive set of views that describe the architecture from a number of viewpoints. For more information, see **#concept# Architectural Views and Viewpoints**.

The architecture can be expressed as a simple metaphor or as a comparison to a predefined architectural style or set of styles. It may be a precise set of models or documents that describe the various aspects of the system's key elements. Expressing it as skeletal implementation is another option - although this may need to be baselined and preserved to ensure that the essence of the system can be understood as the system grows. Choose the medium that best meets the needs of the project.

- **Architectural Patterns**

  Architectural Patterns are ready-made forms that solve recurring architectural problems. An architectural framework or an architectural infrastructure (middleware) is a set of components on which you can build a certain kind of architecture. Many of the major architectural difficulties should be resolved in the framework or in the infrastructure, usually targeted to a specific domain: command and control, MIS, control system, and so on.

  [BUS96] groups architectural patterns according to the characteristics of the systems in which they are most applicable, with one category dealing with more general structuring issues. The table shows the categories presented in [BUS96] and the patterns they contain.

Category	Pattern
Structure	• Layers • Pipes and Filters • Blackboard
Distributed Systems	• Broker
Interactive Systems	• Model-View-Controller • Presentation-Abstraction-Control
Adaptable Systems	• Reflection

	• Microkernel

Refer to [BUS96] for a complete description of these patterns.

- **Architectural Style**

  A software architecture (or an architectural view) may have an attribute called architectural style, which reduces the set of possible forms to choose from, and imposes a certain degree of uniformity to the architecture. The style may be defined by a set of patterns, or by the choice of specific components or connectors as the basic building blocks.

- **Architectural Timing**

  Teams should expect to spend more time on architectural issues early in the project. This allows the team to reduce risk associated to technology early in the project, hence allowing the team to more rapidly reduce the variance in their estimate on what they can deliver at what time. Examples of architectural issues that needs to be resolved early on include the following:

  - **Component** and their major **interfaces.**
  - **Major technology choices** (platform, languages, architecture frameworks / reference architectures, etc.).
  - **Interfaces to external systems.**
  - **Common services** (persistence mechanisms, logging mechanisms, garbage collection, etc.).
  - **Key patterns.**

- **Validating the Architecture**

> The best way to validate the architecture is to actually implement it. For more information, see Executable Architecture.

.

.

#concept# Architectural Constraints

Summary

This concept describes those things that may constrain the architecture of a system.

Main Description

A variety of factors may place constraints on the architecture being developed:

- **Network topology**
- Use of a given **database vendor** or an existing database
- **Web environment** (server configurations, firewall, DMZs, and so forth)
- **Servers** (hardware model, operating system)
- Use of **third-party software** or a **particular technology**
- **Compliance** with existing standards

For example, if the company uses only one type of database, you will probably try to use it as much as possible to leverage the existing database administration skills, rather than introducing a new one.

These architectural **constraints**, combined with the **requirements**, help you define an appropriate candidate for the system architecture. Capturing these constraints will ease integration with the environment; and may **reduce risk, cost and duplication of solution elements**.

.

.

#concept# Architectural Goals

Summary

This concept describes what architectural goals are and why they are important.

Main Description
Architectural goals provide the motivation and rationale for decisions. These goals are often driven by the software requirements, particularly **system-wide requirements** [ALL02].

Architectural goals define how the system needs to respond to change over time. Architectural goals tend to address the following questions:

- What is the expected lifespan of the system?
- Will the system need to respond to technological changes over that time, such as new versions of middleware or other products?
- How frequently is the system expected to adapt to change?
- What changes can we anticipate in the future, and how can we make them easier to accommodate?

These considerations will have a significant effect on the structure of the system.
.

.
#concept# Architecturally Significant Requirements

Summary
This concept describes what architecturally significant requirements are and why they are important.

Main Description
Architecturally significant requirements are those requirements that play an important role in determining the architecture of the system. Such requirements require special attention. Not all requirements have equal significance with regards to the architecture.

Architecturally significant requirements are a subset of the requirements that need to be satisfied before the architecture can be considered "stable". Typically, these are requirements that are **technically challenging**, **technically constraining**, or **central to the system's purpose**.

Furthermore, the system will generally be more sensitive to changes against architecturally significant requirements, so identifying and communicating this subset will help others understand the potential implications of change.

Requirements can be explicitly or implicitly architecturally significant. Explicitly significant requirements are often overtly technical in nature, such as performance targets; the need to interface to other systems; the number of users that must be supported; or security requirements. Implicitly significant requirements may define the essence of the functional behaviour of the system (for example, making a purchase from an on-line store).

Deciding whether a specific requirement is architecturally significant is often a matter of judgment. The selection of requirements that are considered "architecturally significant" is driven by several key driving factors:

- **The benefit of the requirement to stakeholders**: critical, important, or useful.
- **The architectural impact of the requirement**: none, extends, or modifies. There may be critical requirements that have little or no impact on the architecture and low-benefit requirements that have a big impact. Low-benefit requirements with big architectural impacts should be reviewed by the project manager for possible removal from the scope of the project.
- **The risks to be mitigated**: performance, availability of a product, and suitability of a component.
- The completion of the coverage of the architecture.
- Other tactical objectives or constraints, such as demonstration to the user, and so on.

There may be two requirements that hit the same components and address similar risks. If you implement A first, then B is not architecturally significant. If you implement B first, then A is not architecturally significant. Thus these attributes can depend on the order the requirements are realized, and should be re-evaluated when the order changes, as well as when the requirements themselves change.

The following are good examples of Architecturally Significant Requirements:

- The system must record every modification to customer records for audit purposes.
- The system must respond within 5 seconds.
- The system must deploy on Microsoft Windows XP and Linux.
- The system must encrypt all network traffic.
- The ATM system must dispense cash on demand to validated account holders with sufficient cleared funds.

Architecturally significant requirements also describe key behaviors that the system needs to perform. Such **scenarios** represent the important interactions between key abstractions and should be identified as architecturally significant requirements. For example, for an on-line book store describing the way the software handles the scenarios for ordering a book and checking out the shopping cart are often enough to communicate the essence of the architecture.

.

.

#concept# Architectural Views and Viewpoints

Summary

This concept describes the important concepts of views and viewpoints in the context of architecture.

Main Description

Architecture can be represented from a variety of viewpoints, all of which can be combined to create a holistic view of the system. Each architectural view addresses some specific set of concerns, specific to stakeholders in the development process: users, designers, managers, system engineers, maintainers, and so on.

The views capture the major structural design decisions by showing how the software architecture is decomposed into components, and how components are connected by connectors to produce useful forms [PW92]. These design choices must be tied to the requirements -- functional and supplementary -- and other constraints. But these choices in turn put further

constraints on the requirements, and on future design decisions at a lower level.

In essence, architectural views are abstractions, or simplifications, of the entire design, in which **important characteristics** are made more visible by leaving details aside. These characteristics are important when reasoning about:

- **System evolution**-going to the next development cycle.
- **Reuse** of the architecture, or parts of it, in the context of a product line.
- Assessment of **supplementary qualities**, such as performance, availability, portability, and safety.
- Assignment of **development work** to teams or subcontractors.
- Decisions about including off-the-shelf components (@@**buy or make**).
- Insertion in a wider system (@@**Integration with existing systems**).

To choose the appropriate set of views, identify the stakeholders who depend on software architecture documentation and the information that they need. For an example of a set of views that have been used to represent architecture, see **#examples# 4+1 Views of Software Architecture**. A more comprehensive set of views can be found in the IBM Views and Viewpoints Framework for IT systems.

More Information

- **#examples# 4+1 Views of Software Architecture**
-

.

.

#concept# Executable Architecture

Summary

An executable architecture is an implementation that realizes a set of validated architecturally significant requirements.

Main Description

An executable architecture is an implementation that realizes the Software Architecture. It is used to validate that the Architecturally Significant Requirements are correctly implemented. It validates the architecture as an integrated whole through **integration tests**. The team gains feedback about the architecture from the customer or stakeholder by providing the executable architecture for verification. This way the executable architecture helps to assure that the core functionality is stable enough to build the remainder of the system.

An executable architecture is not a work product. It's an identification or attribute of the implementation, indicating that the implementation contains stable architecturally significant functionality.

Each version of an executable architecture should be more complete and robust than previous versions. The final executable architecture contains all the elements that make up the architecture and should validate all architecturally significant requirements. There may be rare exceptions where a portion of the architecture can't practically be implemented until later due to uncontrollable circumstances such as constraints with third part software or unique resources that are unavailable. Delaying any part of the architecture should be avoided as it raises significant technical risk later in the project. But if circumstances dictate that some architectural risk can't be mitigated until later in development, a conscious decision can be made to carry this risk forward until the architecture can be fully implemented.

It's also possible to include non-architectural elements into an executable architecture. This will most likely happen when addressing high risk issues early in the development cycle, which is an excellent practice. Two examples of non-technical risks are resource risks and competitive risks. It may be desirable to obtain a difficult-to-get resource early so they can work on a unique piece of the software now, rather than hoping the resource will be available later. Or it may be useful to implement and deploy some early features to maintain market share against a competitor. Think of the executable architecture as a way to mitigate architectural risk, which is the most significant technical risk in a

project. From this perspective, it's appropriate to mitigate other risks in the executable architecture.

The difference between the executable architecture and the implementation later in the development cycle is that the executable architecture is the result of a period of development (for example an iteration) that's dedicated to elaborating the architecture. Later iterations build onto the executable architecture but are not flagged as an executable architecture because they extend the system's functionality beyond the architectural framework.

More Information

- **#concept# Architecturally Significant Requirements**
- **#concept# Software Architecture**

.

.
#concept# Key Abstractions

Summary
This concept describes key abstractions and the role they play in the architecture

Main Description
Key abstractions are the **key concepts and abstractions** that the system needs to handle. They are those things that, without which, you could not describe the system. (**@@ BCE pattern**)

The requirements are good sources for key abstractions. These abstractions are often easily identified because they represent things that are significant to the business. For example, Customer and Account are typical key abstractions in the banking business.

Each key abstraction should have a short description. The are usually not described in detail as they will change and evolve during the course of the project (as they are refined into actual **design elements**).

The value of defining the key abstractions (and any obvious relationships between them) is that they establish a common understanding of the key concepts amongst the team, thereby enabling them to develop a coherent solution that handles them consistently.

@@similar to Data Entity of Data Architecture in TOGAF.
.

.
#concept# Architectural Mechanism

Summary
Architectural Mechanisms are common solutions to common problems that can be used during development to minimize complexity.

Main Description

- **What are Architectural Mechanisms?**

  Architectural Mechanisms are common solutions to common problems that can be used during development to minimize complexity. They represent **key technical concepts** that will be standardized across the solution. Architecture mechanisms facilitate the evolution of architecturally significant aspects of the system. They allow the team to maintain a cohesive architecture while enabling implementation details to be deferred until they really need to be made.

  Architectural Mechanisms are used to satisfy **architecturally significant requirements**. Usually those are non-functional requirements such as performance and security issues. When fully described, Architectural Mechanisms show patterns of structure and behavior in the software. They form the basis of **common software** that will be consistently applied across the product being developed. They also form the basis for **standardizing** the way that the software works; therefore, they are an important element of the overall software architecture. The definition of architecture mechanisms also enable decisions on whether existing software components can be

leveraged to provide the required behavior; or whether new software should be bought or built (@@buy or build).

The value in defining architecture mechanisms is that they:

- Explicitly call out aspects of the solution mechanics that are common across the system. This helps you plan.
- Put down markers for the developers to build those aspects of the system once and then re-use them. This reduces the workload.
- Promote the development of a consistent set of services. This makes the system easier to maintain.

An Architectural Mechanism can have three states: **Analysis, Design** and **Implementation**. These categories reflect the maturity of the mechanism's description. The state changes as successive levels of detail are uncovered during when you refine Architecturally Significant Requirements into working software. The categories are summarized in the table that follows.

States of an Architectural Mechanism

State	Description
Analysis	A conceptual solution to a common technical problem. For example, **persistence** is an abstract solution to the common requirement to store data. The purpose of this category is simply to identify the need for an Architectural Mechanism to be designed and implemented; and capture basic attributes for that mechanism.
Design	A refinement of an Analysis Mechanism into a **concrete technology** (for example, RDBMS). The purpose of this category is to guide precise product or technology selection.
Implementation	A further refinement from a design

	mechanism into a **specification for the software**. This can be presented as a design pattern or example code.

For more information on these different types of mechanisms, see the attached concepts.

Be aware that these states are frequently referred to themselves as Analysis, Design and Implementation mechanisms. These are synonyms and merely represent the architecture mechanisms in different states of development. The transition from one state to another can often be obvious or intuitive. Therefore, it can be achieved in a matter of seconds. It can also require more considered analysis and design, thus take longer. The important point here is that these categories of mechanisms apply to the same concept in different states. The only difference between them is one of **refinement** or **detail**.

The following diagram illustrates the transition of Architectural Mechanisms from one state to another.

- **What Information Should be Captured for Architectural Mechanisms?**

  The information captured for each architectural mechanism category/state is different (though the information can be seen as refinements of each other):

  - **Analysis Mechanisms**, which give the mechanism a name, brief description and some basic attributes derived from the project requirements.
  - **Design Mechanisms**, which are more concrete and assume some details of the implementation environment.
  - **Implementation Mechanisms**, which specify the exact implementation of each mechanism.

When a mechanism is initially identified, it can be considered a marker that says to the team, "We are going to handle this aspect of the system in a standard way. We'll figure out the details later." As the project proceeds, the architectural mechanisms are gradually refined until they become part of the software.

- **Analysis Mechanisms**

  Analysis mechanisms are the initial state for an architectural mechanism. They are identified early in the project and represent bookmarks for future software development. They allow the team to focus on understanding the requirements without getting distracted by the specifics of a complex implementation. Analysis mechanisms are discovered by surveying the requirements and looking for recurrent technical concepts. **Security**, **persistence** and **legacy interface** are some examples of these. In effect, the analysis mechanism is where the requirements that describe architecturally significant topics are collated and brought together in a single list. This makes them easier to manage.

  Analysis mechanisms are described in simple terms:

  - **Name**: Identifies the mechanism.
  - **Basic attributes**: Define the requirements of the mechanism. These attributes can vary depending upon the mechanism being analyzed. Refer to Example: Architectural Mechanism Attributes for more guidance.

  Once the list of analysis mechanisms has been defined it can be prioritized and the mechanisms refined in line with iteration objectives. It is not necessary to develop the entire set of architecture mechanisms into working software in a single pass. It is often more sensible to develop only those mechanisms required to support the functionality to be delivered in the current iteration.

- **Design Mechanisms**

  Design mechanisms represent decisions about the **concrete technologies** that are going to be used to develop architectural mechanisms. For example, the decision to use an **RDBMS** for persistence. It's often no more complicated than that (though of course, the effort involved in making the decision can sometimes be quite complex).

  The decision on when to refine an architectural mechanism from an analysis state to a design state is largely arbitrary. Often there will be constraints on the project that force the decision on some of these issues. For example, there may be a corporate standard for databases which mean that the decision for the persistence mechanism can be made on day 1 of the project.

  On other occasions the decision may point to products that the project team has not yet acquired. If so, the decision needs to be made in time to enable the required products to be made available to the team.

  It can often be useful to develop some prototype code to prove that these decisions are sound. **The architect should be confident that the technologies being selected are able to fulfill the requirements**. The attributes captured against the corresponding analysis mechanisms should be used as criteria to prove the validity of the decisions.

- **Implementation Mechanism**

  An implementation mechanism specifies the **actual implementation** for the architectural mechanism (hence the name). It can be modeled as a design pattern or presented as example code.

  The best time to produce the implementation mechanism is usually when the first piece of functionality that needs it is scheduled for development. Architects and developers work together to develop this.

For examples of the kinds of information that you might capture for a mechanism, see **#example# Architectural Mechanism Attributes.**

More Information

- **#concept#** Analysis Mechanism
- **#concept#** Architecturally Significant Requirements
- **#concept#** Design Mechanism
- **#concept#** Implementation Mechanism
- **#concept#** Pattern
- **#example#** Examples
- **#example#** Architectural Mechanism Attributes
- **#guideline#** Example: Design Mechanisms
- **#guideline#** Identify Common Architectural Mechanisms

.

.

#concept# Analysis Mechanism

Summary

An Analysis Mechanism is a conceptual representation of an Architectural Mechanism.

Main Description

An Analysis Mechanism is a **conceptual representation** of an Architectural Mechanism. Over time, **Analysis Mechanisms** are refined into **Design Mechanisms** and, later, into **Implementation Mechanisms**.

Analysis Mechanisms allow the developer to focus on understanding the requirements without getting distracted by the specifics of a complex implementation. They are a way of abstracting away the complexity of the solution, so people can better comprehend the problem.

Analysis Mechanisms are described in simple terms:

- **Name:** Identifies the mechanism.

- **Basic attributes:** Define the requirements of the mechanism.

You can identify Analysis Mechanisms top-down, from previous knowledge, or bottom-up, meaning that you discover them as you proceed.

In the **top-down mode**, you are guided by experience -- you know that certain problems are present in the domain and will require certain kinds of solutions. Examples of common architectural problems that might be expressed as mechanisms during analysis are:

- **persistence,**
- **transaction management,**
- **fault management,**
- **messaging,**
- **inference engines**.

The common aspect of all of these is that each is a general capability of a broad class of systems, and each provides functionality that interacts with or supports the basic application functionality. The Analysis Mechanisms support capabilities required in the basic functional requirements of the system, regardless of the platform that it is deployed upon or the implementation language. Analysis Mechanisms also can be designed and implemented in different ways. Generally, there will be more than one design mechanism that corresponds with each Analysis Mechanism. There may also be more than one way of implementing each design mechanism.

The bottom-up approach is where Analysis Mechanisms ultimately originate. They are created as the you see, perhaps faintly at first, a common theme emerging from a set of solutions to various problems. For example: There is a need to provide a way for elements in different threads to synchronize their clocks, and there is a need for a common way of allocating resources. Analysis Mechanisms, which

simplify the language of analysis, emerge from these patterns.

Identifying an Analysis Mechanism means that you identify a common, perhaps implicit subproblem, and you give it a name. Initially, the name might be all that exists. For example, the system will require a persistence mechanism. Ultimately, this mechanism will be implemented through the collaboration of various classes, some of which do not deliver application functionality directly, but exist only to support it. Very often these support classes are located in the middle or lower layers of a layered architecture, thereby providing a common support service to all application-level classes.

If the subproblem that you identify is common enough, perhaps a pattern exists from which the mechanism can be instantiated, probably by binding existing classes and implementing new ones, as required by the pattern. An Analysis Mechanism produced this way will be abstract, and it will require further refinement throughout design and implementation work.

You can see examples of how Architectural Mechanisms can be represented in Example: Architectural Mechanism Attributes.
Back to top

More Information

- **#concept# Architectural Mechanism**
- **#concept# Design Mechanism**
- **#concept# Implementation Mechanism**
- **#example# Architectural Mechanism Attributes**

.

.
#concept# Design Mechanism

Summary
A Design Mechanism is a concrete representation of an Architectural Mechanism.

Main Description

A Design Mechanism is a concrete representation of an Architectural Mechanism. It is refined from an Analysis Mechanism and is further refined into an Implementation Mechanism as the design becomes more detailed.

Design Mechanisms can be represented as specific **design patterns** and **frameworks** in the design. They are used to guide development. Design Mechanisms should still be relatively independent of implementation but provide enough detailed information for implementation choices to be made and software to be developed with confidence.

See **#guideline# Example: Design Mechanisms.**

More Information

- **#concept# Analysis Mechanism**
- **#concept# Architectural Mechanism**
- **#concept# Implementation Mechanism**
- **#guideline# Example: Design Mechanisms**

@@We could use the platform service of TRM as the cadinates of design mechanism.
.

.
#concept# Implementation Mechanism

Summary
A representation of an Architecture Mechanism that uses a specific programming language or product.

Main Description
An Implementation Mechanism is a refinement of a corresponding Design Mechanism that uses, for example, a particular programming language and other implementation technology (such as a particular vendor's middleware product). An Implementation Mechanism may instantiate one or more idioms or implementation patterns.

Review these points when you are considering Implementation Mechanisms:

- **Determine the ranges of characteristics.** Take the characteristics that you identified for the Design Mechanisms into consideration to determine reasonable, economical, or feasible ranges of values to use in the Implementation Mechanism candidate.
- **Consider the cost of purchased components.** For Implementation Mechanism candidates, consider the cost of licensing, the maturity of the product, your history or relationship with the vendor, support, and so forth in addition to purely technical criteria.
- **Conduct a search for the right components, or build the components.** You will often find that there is no apparently suitable Implementation Mechanism for a particular Design Mechanism. This will either trigger a search for the right product or make the need for in-house development apparent. You may also find that some Implementation Mechanisms are not used at all.

The choice of Implementation Mechanisms is based not only on a good match for the technical characteristics, but also on the non-technical characteristics, such as cost. Some of the choices may be provisional. Almost all have some risks attached to them. **Performance**, **robustness**, and **scalability** are nearly always concerns and must be validated by evaluation, exploratory prototyping, or inclusion in the architectural prototype.