UNIT-II

Analyzing Architectures

One of the most important truths about the architecture of a system is that knowing it will tell you important properties of the system itself?even if the system does not yet exist. Architects make design decisions because of the downstream effects they will have on the system(s) they are building, and these effects are known and predictable. If they were not, the process of crafting an architecture would be no better than throwing dice: We would pick an architecture at random, build a system from it, see if the system had the desired properties, and go back to the drawing board if not. While architecture is not yet a cookbook science, we know we can do much better than randomguessing.

Architects by and large know the effects their design decisions will have. As we saw in

Chapter 5, architectural tactics and patterns in particular bring known properties to the systems in which they are used. Hence, design choices?that is to say, architectures?are analyzable. Given an architecture, we can deduce things about the system, even if it has not been built yet.Why evaluate an architecture? Because so much is riding on it, and because you can. An effective technique to assess a candidate architecture?*before*it becomes the project's accepted blueprint?is of great economic value. With the advent of repeatable, structured methods (such as the ATAM, presented in Chapter 11), architecture evaluation has come to provide relatively a low-cost risk mitigation capability. Making sure the architecture is the right one simply makes good sense. *An architecture evaluation should be a standard part of every architecture-based development methodology*.

It is almost always cost-effective to evaluate software quality as early as possible in the life cycle. If problems are found early, they are easier to correct?a change to a requirement, specification, or design is all that is necessary. Software quality cannot be appended late in a project, but must be inherent from the beginning, built in by design. It is in the project's best interest for prospective candidate designs to be evaluated (and rejected, if necessary) during the design phase, before long-term institutionalization.

However, architecture evaluation can be carried out at many points during a system's life cycle. If the architecture is still embryonic, you can evaluate those decisions that have already been made or are being considered. You can choose among architectural alternatives.

If the architecture is finished, or nearly so, you can validate it before the project commits to lengthy and expensive development. It also makes sense to evaluate the architecture of a legacy system that is undergoing modification, porting, integration with other systems, or other significant upgrades. Finally, architecture evaluation makes an

excellent discovery vehicle: Development projects often need to understand how an inherited system meets (or whether it meets) its quality attribute requirements.

Furthermore, when *acquiring* a large software system that will have a long lifetime, it is important that the acquiring organization develop an understanding of the underlying architecture of the candidate. This makes an assessment of their suitability possible with respect to qualities of importance.Evaluation can also be used to choose between two competing architectures by evaluating both and seeing which one fares better against the criteria for "goodness."

We enumerate six benefits that flow from holding architectural inspections.

1. *Financial*. At AT&T, each project manager reports perceived savings from an architecture evaluation. On average, over an eight-year period, projects receiving a full architecture evaluation have reported a 10% reduction in project costs. Given the cost estimate of 70 staff-days, this illustrates that on projects of 700 staff-days or longer the review pays for itself.

   Other organizations have not publicized such strongly quantified data, but several consultants have reported that more than 80% of their work was repeat business. Their customers recognized sufficient value to be willing to pay for additional evaluations.

   There are many anecdotes about estimated cost savings for customers' evaluations. A large company avoided a multi-million-dollar purchase when the architecture of the global information system they were procuring was found to be incapable of providing the desired system attributes. Early architectural analysis of an electronic funds transfer system showed a $50 billion transfer capability per night, which was only half of the desired capacity. An evaluation of a retail merchandise system revealed early that there would be peak order performance problems that no amount of hardware could fix, and a major business failure was prevented. And so on.

There are also anecdotes of architecture evaluations that did not occur but should have. In one, a rewrite of a customer accounting system was estimated to take two years but after seven years the system had been reimplemented three times. Performance goals were never met despite the fact that the latest version used sixty times the CPU power of the original prototype version. In another case, involving a large engineering relational database system, performance problems were largely attributable to design

decisions that made integration testing impossible. The project was canceled after $20 million had been spent.

2. *Forced preparation for the review.* Indicating to the reviewees the focus of the architecture evaluation and requiring a representation of the architecture before the evaluation is done means that reviewees must document the system's architecture. Many systems do not have an architecture that is understandable to all developers. The existing description is either too brief or (more commonly) too long, perhaps thousands of pages. Furthermore, there are often misunderstandings among developers about some of the assumptions for their elements. The process of preparing for the evaluation will reveal many of theseproblems.

3. *Captured rationale.* Architecture evaluation focuses on a few specific areas with specific questions to be answered. Answering these questions usually involves explaining the design choices and their rationales. A documented design rationale is important later in the life cycle so that the implications of modifications can be assessed. Capturing a rationale after the fact is one of the more difficult tasks in software development. Capturing it as presented in the architecture evaluation makes invaluable information available for lateruse.

4. *Early detection of problems with the existing architecture.* The earlier in the life cycle that problems are detected, the cheaper it is to fix them. The problems that can be found by an architectural evaluation include unreasonable (or expensive) requirements, performance problems, and problems associated with potential downstream modifications. An architecture evaluation that exercises system modification scenarios can, for example, reveal portability and extensibility problems. In this way an architecture evaluation provides early insight into product capabilities andlimitations.
5. *Validation of requirements.* Discussion and examination of how well an architecture meets requirements opens up the requirements for discussion. What results is a much clearer understanding of the requirements and, usually, their prioritization. Requirements creation, when isolated from early design, usually results in conflicting system properties. High performance, security, fault tolerance, and low cost are all easy to demand but difficult to achieve, and often impossible to achieve simultaneously. Architecture evaluations uncover the conflicts and tradeoffs, and provide a forum for their*negotiated*resolution.
6. *Improved architectures*. Organizations that practice architecture evaluation as a standard part of their development process report an improvement in the quality ofthe

Evaluations can be planned or unplanned. A planned evaluation is considered a normal part of the project's development cycle. It is scheduled well in advance, built into the project's work plans and budget, and follow-up is expected. An unplanned evaluation is unexpected and usually the result of a project in serious trouble and taking extreme measures to try to salvage previous effort.

The planned evaluation is ideally considered an asset to the project, at worst a distraction from it. It can be perceived not as a challenge to the technical authority of the project's members but as a validation of the project's initial direction. Planned evaluations are pro- active and team-building.

An unplanned evaluation is more of an ordeal for project members, consuming extra project resources and time in the schedule from a project already struggling with both. It is initiated only when management perceives that a project has a substantial possibility of failure and needs to make a mid-course correction. Unplanned evaluations are reactive, and tend to be tension filled. An evaluation's team leader must take care not to let the activities devolve into finger pointing.

Needless to say, planned evaluations are preferable.

A successful evaluation will have the following properties:

1. *Clearly articulated goals and requirements for the architecture.* An architecture is only suitable, or not, in the presence of specific quality attributes. One that delivers breathtaking performance may be totally wrong for an application that needs modifiability. Analyzing an architecture without knowing the exact criteria for "goodness" is like beginning a trip without a destination in mind. Sometimes (but in ourexperience,

almost never), the criteria are established in a requirements specification. More likely, they are elicited as a precursor to or as part of the actual evaluation. Goals define the purpose of the evaluation and should be made an explicit portion of the evaluation contract, discussed subsequentl

.

2. *Controlled scope.* In order to focus the evaluation, a small number of explicit goals should be enumerated. The number should be kept to a minimum?around three to five?an inability to define a small number of high-priority goals is an indication that the expectations for the evaluation (and perhaps the system) may beunrealistic.
3. *Cost-effectiveness.* Evaluation sponsors should make sure that the benefits of the evaluation are likely to exceed the cost. The types of evaluation we describe are suitable for medium and large-scale projects but may not be cost-effective for smallprojects.
4. *Key personnel availability.* It is imperative to secure the time of the architect or at least someone who can speak authoritatively about the system's architecture and design. This person (or these people) primarily should be able to communicate the facts of the architecture quickly and clearly as well as the motivation behind the architectural decisions. For very large systems, the designers for each major component need to be involved to ensure that the architect's notion of the system design is in fact reflected and manifested in its more detailed levels. These designers will also be able to speak to the behavioral and quality attributes of the components. For the ATAM, the architecture's stakeholders need to be identified and represented at the evaluation. It is essential to identify the customer(s) for the evaluation report and to elicit their values and expectations.
5. *Competent evaluation team.* Ideally, software architecture evaluation teams are separate entities within a corporation, and must be perceived as impartial, objective, and respected. The team must be seen as being composed of people appropriate to carry out the evaluation, so that the project personnel will not regard the evaluation as a waste of time and so that its conclusions will carry weight. It must include people fluent in architecture and architectural issues and be led by someone with solid experience in designing and evaluating projects at the architecturallevel.
6. *Managed expectations.* Critical to the evaluation's success is a clear, mutual understanding of the expectations of the organization sponsoring it. The evaluation should be clear about what its goals are, what it will produce, what areas it will (and will not) investigate, how much time and resources it will take from the project, and to whom the results will bedelivered.

The ATAM: A Comprehensive Method for Architecture Evaluation

we will introduce the Architecture Tradeoff Analysis Method (ATAM), a thorough and comprehensive way to evaluate a software architecture. The ATAM is so named because it reveals how well an architecture satisfies particular quality goals, and (because it recognizes that architectural decisions tend to affect more than one quality attribute) it provides insight into how quality goals interact?that is, how they trade off.

The ATAM is designed to elicit the business goals for the system as well as for the architecture. It is also designed to use those goals and stakeholder participation to focus the attention of the evaluators on the portion of the architecture that is central to the achievement of the goals.

Participants in the ATAM

The ATAM requires the participation and mutual cooperation of three groups:

1. *The evaluation team.* This group is external to the project whose architecture is being evaluated. It usually consists of three to five people. Each member of the team is assigned a number of specific roles to play during the evaluation. (See Table 11.1 for a description of these roles, along with a set of desirable characteristics for each.) The evaluation team may be a standing unit in which architecture evaluations are regularly performed, or its members may be chosen from a pool of architecturally savvy individuals for the occasion. They may work for the same organization as the development team whose architecture is on the table, or they may be outside consultants. In any case, they need to be recognized as competent, unbiased outsiders with no hidden agendas or axes togrind.
2. *Project decision makers.* These people are empowered to speak for the development project or have the authority to mandate changes to it. They usually include the project manager, and, if there is an identifiable customer who is footing the bill for the development, he or she will be present (or represented) as well. The architect is always included?a cardinal rule of architecture evaluation is that the architect must willingly participate. Finally, the person commissioning the evaluation is usually empowered to speak for the development project; even if not, he or she should be included in the group.
3. *Architecture stakeholders.* Stakeholders have a vested interest in the architecture performing as advertised. They are the ones whose ability to do their jobs hinges onthe

architecture promoting modifiability, security, high reliability, or the like. Stakeholders include developers, testers, integrators, maintainers, performance engineers, users, builders of systems interacting with the one under consideration, and others. Their job during an evaluation is to articulate the specific quality attribute goals that the architecture should meet in order for the system to be considered a success. A rule of thumb?and that is all it is?is that you should expect to enlist the services of twelve to fifteen stakeholders for the evaluation.

Outputs of the ATAM

An ATAM-based evaluation will produce at least the following outputs:

- *A concise presentation of the architecture.* Architecture documentation is often thought to consist of the object model, a list of interfaces and their signatures, or some other voluminous list. But one of the requirements of the ATAM is that the architecture be presented in one hour, which leads to an architectural presentation that is both concise and, usually,understandable.
- *Articulation of the business goals.* Frequently, the business goals presented in the ATAM are being seen by some of the development team for the firsttime.
- *Quality requirements in terms of a collection of scenarios.* Business goals lead to quality requirements. Some of the important quality requirements are captured in the form of scenarios.
- *Mapping of architectural decisions to quality requirements.* Architectural decisions can be interpreted in terms of the qualities that they support or hinder. For each quality scenario examined during an ATAM, those architectural decisions that help to achieve it are determined.
- *A set of identified sensitivity and tradeoff points.* These are architectural decisions that have a marked effect on one or more quality attributes. Adopting a backup database, for example, is clearly an architectural decision as it affects reliability (positively), and so it is a sensitivity point with respect to reliability. However, keeping the backup current consumes system resources and so affects performance negatively. Hence, it is a tradeoff point between reliability and performance. *A set of risks and nonrisks.* A risk is defined in the ATAM as an architectural decision that may lead to undesirable consequences in light of stated quality attributerequirements.

Similarly, a nonrisk is an architectural decision that, upon analysis, is deemed safe. The identified risks can form the basis for an architectural risk mitigation plan.

- *A set of risk themes.* When the analysis is complete, the evaluation team will examine the full set of discovered risks to look for over-arching themes that identify systemic weaknesses in the architecture or even in the architecture process and team. If left untreated, these risk themes will threaten the project's businessgoals.

Phases of the ATAM

ATAM Phases and Their Characteristics

Phase	Activity	Participants	TypicalDuration
0	Partnership and preparation	Evaluation team leadership and key project decision makers	Proceeds informally as required, perhaps over a few weeks
1	Evaluation	Evaluation team andproject decision makers	1 day followed by a hiatus of 2 to 3weeks
2	Evaluation (continued)	Evaluation team, project decision makers, and stakeholders	2 days
3	Follow-up	Evaluation team and evaluation client	1 week

The CBAM: A Quantitative Approach to Architecture Design Decision Making

the ATAM is missing an important consideration: The biggest tradeoffs in large, complex systems usually have to do with economics. How should an organization invest its resources in a manner that will maximize its gains and minimize its risk? In the past, this question primarily focused on costs, and even then these were primarily the costs of building the system in the first place and not the long-term costs through cycles of maintenance and

upgrade. As important, or perhaps more important than costs, are the *benefits*that an architectural decision may bring to an organization.

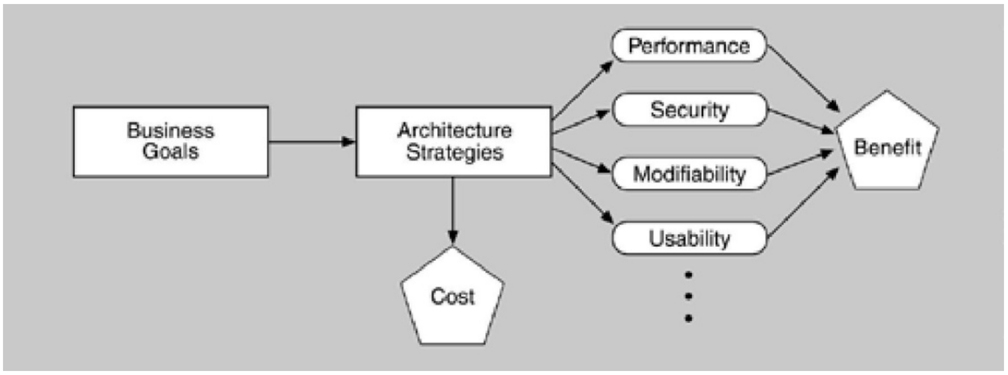
Given that the resources for building and maintaining a system are finite, there must be a rational process that helps us choose among architectural options, during both an initial design phase and subsequent upgrade periods. These options will have different costs, will consume differing amounts of resources, will implement different features (each of which brings some benefit to the organization), and will have some inherent risk or uncertainty. To capture these aspects we need *economic* models of software that take into account costs, benefits, risks, and schedule implications.

To address this need for economic decision making, we have developed a method of economic modeling of software systems, centered on an analysis of their architectures. Called the Cost Benefit Analysis Method (CBAM), it builds on the ATAM to model the costs and the benefits of architectural design decisions and is a means of optimizing such decisions. The CBAM provides an assessment of the technical and economic issues and architectural decisions.

Decision-Making Context

The software architect or decision maker wishes to maximize the difference between the benefit derived from the system and the cost of implementing the design. The CBAM begins where the ATAM concludes and, in fact, depends upon the artifacts that the ATAM produces as output. Figure 12.1 depicts the context for the CBAM.

Context for the CBAM



The Basis for theCBAM

UTILITY

Utility is determined by considering the issues described in the following sections.

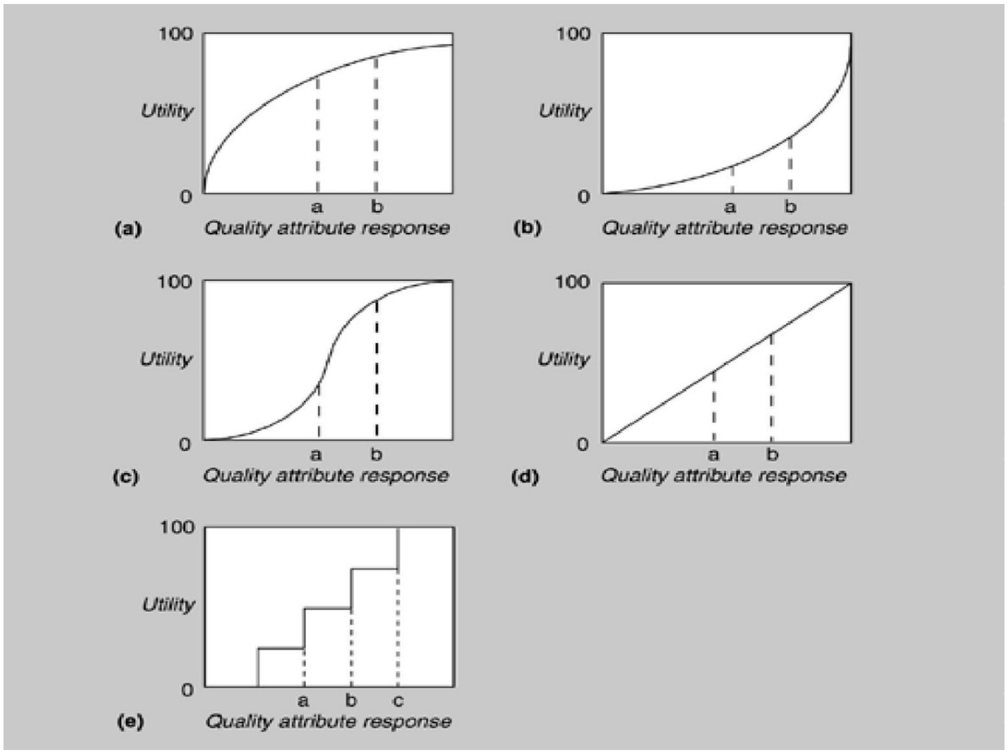
Variations of Scenarios

The CBAM uses scenarios as a way to concretely express and represent specific quality attributes, just as in the ATAM. Also as in the ATAM, we structure scenarios into three parts: stimulus (an interaction with the system), environment (the system's state at the time), and response (the measurable quality attribute that results). However, there is a difference between the methods: The CBAM actually uses a *set* of scenarios (generated by varying the values of the responses) rather than individual scenarios as in the ATAM. This leads to the concept of a utility-responsecurve.

Utility-Response Curves

Every stimulus-response value pair in a scenario provides some utility to the stakeholders, and the utility of different possible values for the response can be compared. For example, a very high availability in response to failure might be valued by the stakeholders only slightly more than moderate availability. But low latency might be valued substantially more than moderate latency. We can portray each relationship between a set of utility measures and a corresponding set of response measures as a graph?a utility-response curve. Some examples of utility-response curves are shown in Figure 12.2. In each, points labeled a, b, or c represent different response values. The utility-response curve thus shows utility as a function of the responsevalue.

Figure 12.2. Some sample utility-response curves



The utility-response curve depicts how the utility derived from a particular response varies as the response varies. As seen in Figure 12.2, the utility could vary nonlinearly, linearly, or even as a step-function. For example, graph (c) portrays a steep rise in utility over a narrow change in a quality attribute response level, such as the performance example stated above. The availability example might be better characterized by graph (a), where a modest change in the response level results in only a very small change in utility to theuser.

Priorities of Scenarios

Architectural Strategies

Side

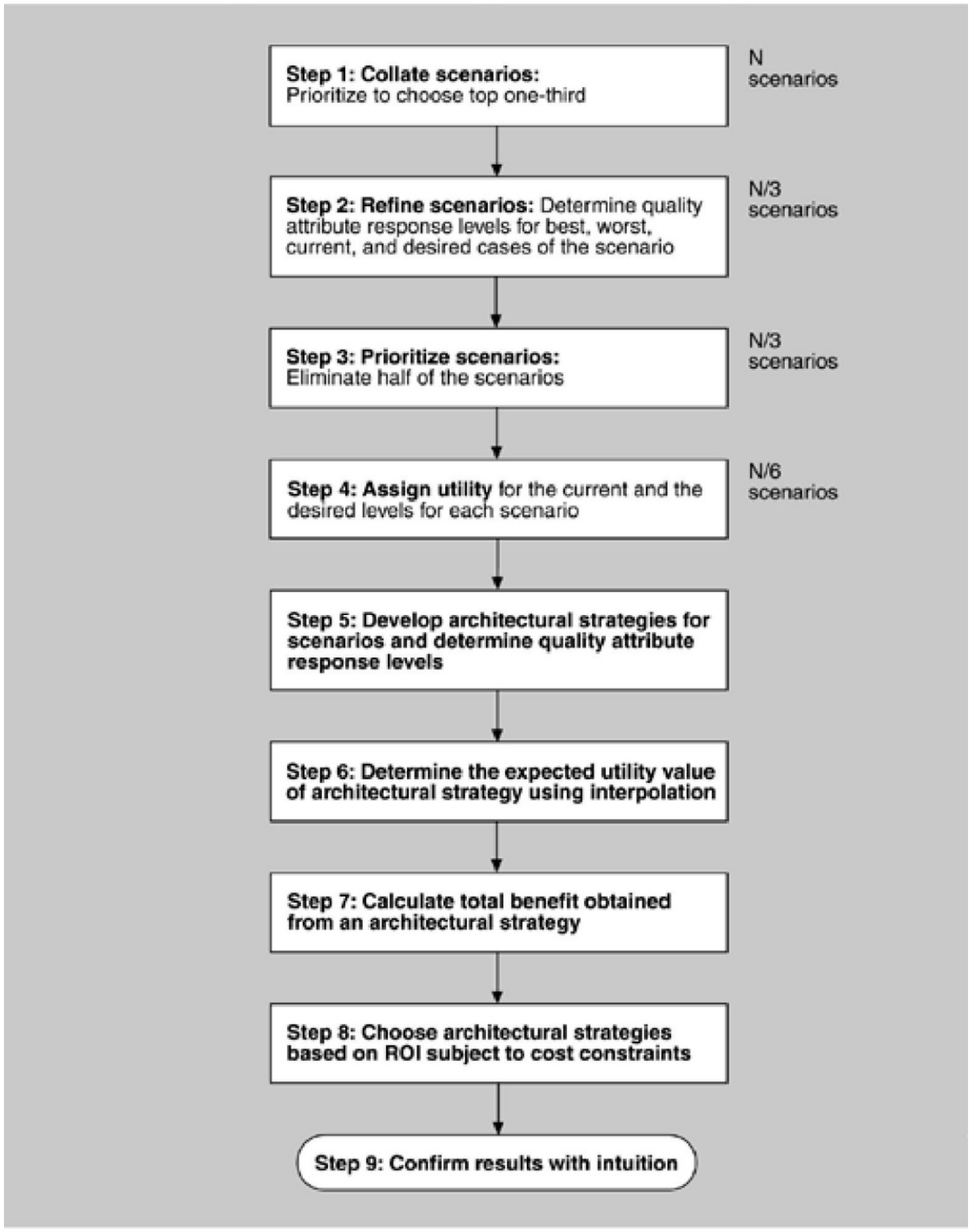
effectsDetermining

benefit and

normalization

Implementing the CBAM

Process flow diagram for the CBAM


Step 1: Collate scenarios: Prioritize to choose top one-third
N scenarios
Step 2: Refine scenarios: Determine quality attribute response levels for best, worst, current, and desired cases of the scenario
N/3 scenarios
Step 3: Prioritize scenarios: Eliminate half of the scenarios
N/3 scenarios
Step 4: Assign utility for the current and the desired levels for each scenario
N/6 scenarios
Step 5: Develop architectural strategies for scenarios and determine quality attribute response levels
Step 6: Determine the expected utility value of architectural strategy using interpolation
Step 7: Calculate total benefit obtained from an architectural strategy
Step 8: Choose architectural strategies based on ROI subject to cost constraints
Step 9: Confirm results with intuition

MovingFromOneSystemtoMany

focuses on the construction of multiple systems from that architecture, discussing, and giving examples of system product lines. It does this from five perspectives: that of the technology underlying a product line, that of a single company that built a product line of naval vessel fire-control systems, that of an industry-wide architecture, that of a single company producing products based on the industry-wide architecture, and that of an organization building systems from commercial components.

Software Product Lines: Re-using Architectural Assets

A software architecture represents a significant investment of time and effort, usually by senior talent. So it is natural to want to maximize the return on this investment by re-using an architecture across multiple systems. Architecturally mature organizations tend to treat their architectures as valuable intellectual property and look for ways in which that property can be leveraged to produce additional revenue and reduce costs. Both are possible with architecture re-use.

Software product lines based on inter-product commonality represent an innovative, growing concept in software engineering. Every customer has its own requirements, which

demand flexibility on the part of the manufacturers. Software product lines simplify the creation of systems built specifically for particular customers or customer groups.

What Makes Software Product Lines Work?

The essence of a software product line is the disciplined, strategic re-use of assets in producing a family of products. What makes product lines succeed so spectacularly from the vendor or developer's point of view is that the commonalities shared by the products can be exploited through re-use to achieve production economies. The potential for re-use is broad and far-ranging, including:

- *Requirements.* Most of the requirements are common with those of earlier systems and so can be re-used. Requirements analysis issaved.
- *Architectural design.* An architecture for a software system represents a large investment of time from the organization's most talented engineers. As we have seen, the quality goals for a system?performance, reliability, modifiability, and so forth?are largely allowed or precluded once the architecture is in place. If the architectureis

Wrong, the system cannot be saved. For a new product, however, this most important design step is already done and need not be repeated.

- *Elements.* Software elements are applicable across individual products. Far and above mere code re-use, element re-use includes the (often difficult) initial design work. Design successes are captured and re-used; design dead ends are avoided, not repeated. This includes design of the element's interface, its documentation, its test

  which represents an enormous and vital set of design decisions.

- *Modeling and analysis.* Performance models, schedulability analysis, distributedsystem

- plans and procedures, and any models (such as performance models) used to predict or measure its behavior. One re-usable set of elements is the system's userinterface,

  issues (such as proving absence of deadlock), allocation of processes to processors, fault tolerance schemes, and network load policies all carry over from product to product. CelsiusTech (as discussed in Chapter 15) reports that one of the major headaches associated with the real-time distributed systems it builds has all but vanished. When fielding a new product in the product line, it has extremely high confidence that the timing problems have been worked out and that **the bugs** associated with distributed

  computing?synchronization, network loading, deadlock?have beeneliminated.

- *Testing.* Test plans, test processes, test cases, test data, test harnesses, andthe communication paths required to report and fix problems are already inplace.

- *Project planning.* Budgeting and scheduling are more predictable because experienceis a high-fidelity indicator of future performance. Work breakdown structures need not be invented each time. Teams, team size, and team composition are all easily determined.

- *Processes, methods, and tools.* Configuration control procedures and facilities, documentation plans and approval processes, tool environments, system generationand distribution procedures, coding standards, and many other day-to-day engineering support activities can all be carried over from product to product. The overallsoftware development process is in place and has been used before.

- *People.* Because of the commonality of applications, personnel can be fluidly transferred among projects as required. Their expertise is applicable across the entireline.

- *Exemplar systems.* Deployed products serve as high-quality demonstration prototypes as well as high-quality engineering models of performance, security, safety, and reliability.

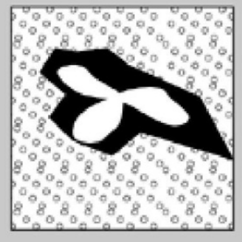
- *Defect elimination.* Product lines enhance quality because each new system takes advantage of the defect elimination in its forebears. Developer and customer confidence both rise with each new instantiation. The more complicated the system, the higher the payoff for solving vexing performance, distribution, reliability, and other engineering issues once for the entirefamily.

Scoping

The scope of a product line defines what systems are in it, and what systems are out. Put less bluntly, a product line's scope is a statement about what systems an organization is willing to build as part of its line and what systems it is not willing to build. Defining a product line's scope is like drawing a doughnut in the space of all possible systems, as shown in Figure 14.1. The doughnut's center represents the systems that the organization could build, would build, because they fall within its product line capability. Systems outside the doughnut are out of scope, ones that the product line is not well equipped to handle. Systems on the doughnut itself could be handled, but with some effort, and require case- by-case disposition as they arise. To illustrate, in a product line of office automation systems a conference room scheduler would be in; a flight simulator would be out. A specialized intranet search engine might be in if it could be produced in a reasonable time and if there were strategic reasons for doing so (such as the likelihood that future customers would want a similar product).

Figure 14.1. The space of all possible systems is divided into areas within scope (*white*), areas outside of scope (*speckled*), and areas that require case-by-case disposition (*black*).

Adapted from [Clements 02b].



The scope represents the organization's best prediction about what products it will be asked to build in the foreseeable future. Input to the scoping process comes from the organization's strategic planners, marketing staff, domain analysts who can catalog similar systems (both existing and on the drawing board), and technology experts.

Architectures for Product Lines

Of all of the assets in a core asset repository, the software architecture plays the most central role. The essence of building a successful software product line is discriminating between what is expected to remain constant across all family members and what is expected to vary. Software architecture is ready-made for handling this duality, since all architectures are abstractions that admit a plurality of instances; a great source of their conceptual value is, after all, that they allow us to concentrate on design essentials within a number of different implementations. By

about what we expect to remain constant and what we admit may vary. In a software product line, the architecture is an expression of the nonvarying aspects.

But a product line architecture goes beyond this simple dichotomy, concerning itself with a

its very nature an architecture is a statement

set of explicitly allowed variations, whereas with a conventional architecture almost any instance will do as long as the (single) system's behavioral and quality goals are met. Thus, identifying the allowable variations is part of the architecture's responsibility, as is providing built-in mechanisms for achieving them. Those variations may be substantial. Products in a software product line exist simultaneously and may vary in terms of their behavior, quality attributes, platform, network, physical configuration, middleware, scale factors, and so forth.

A product line architect needs to consider three things:

- Identifying variationpoints
- Supporting variationpoints
- Evaluating the architecture for product linesuitability

What Makes Software Product Lines Difficult?

It takes a certain maturity in the developing organization to successfully field a product line. Technology is not the only barrier to this; organization, process, and business issues are equally vital to master to fully reap the benefits of the software product line approach.

The Software Engineering Institute has identified twenty-nine issues or "practice areas" that affect an organization's success in fielding a software product line. Most of these practice areas are applied during single-system development as well, but take on a new dimension in a product line context. Two examples are architecture definition and configuration management.

ADOPTION STRATEGIES

CREATING PRODUCTS AND EVOLVING A PRODUCT LINE

External sources

Internal sources

ORGANIZATIONAL STRUCTURE

Development department

Business units.

Domain engineering unit

Hierarchical domain engineering units

Building Systems from Off-the-Shelf Components

Operating systems impose certain solutions and have since the 1960s. Database management systems have been around since the early 1970s. Because of the ubiquity of computers the possibility of using externally developed components to achieve some system goals has been increasing dramatically. Even the availability of components may not cause you to use or keep them (see the sidebar Quack.com), but you certainly need to understand how to incorporate them into your system.

For systems built from off-the-shelf (OTS) components, component selection involves a discovery process, which seeks to identify *assemblies* of compatible components, understanding how they can achieve the desired quality attributes, and deciding whether they can be integrated into the system being built.

Impact of Components onArchitecture

Consider the following situation. You are producing software to control a chemical plant. Within chemical plants, specialized displays keep the operator informed as to the state of the reactions being controlled. A large portion of the software you are constructing is used to draw those displays. A vendor sells user interface controls that produce them. Because it is easier to buy than build, you decide to purchase the controls?which, by the way, are only available for Visual Basic.

What impact does this decision have on your architecture? Either the whole system must be written in Visual Basic with its built-in callback-centered style or the operator portion must be isolated from the rest of the system in some fashion. This is a fundamental structural decision, driven by the choice of a single component for a single portion of the system.

The use of off-the-shelf components in software development, while essential in many cases, also introduces new challenges. In particular, component capabilities and liabilities are a principle architectural constraint.

Architectural Mismatch

Architectural mismatch is a special case of *interface mismatch*, where the interface is as Parnas defined it: the assumptions that components can make about each other. This definition goes beyond what has, unfortunately, become the standard concept of interface in current practice: a component's API (for example, a Java interface specification). An API names the programs and their parameters and may say something about their behavior, but this is only a small part of the information needed to correctly use a component. Side effects, consumption of global resources, coordination requirements, and the like, are aInterface mismatch can appear at integration time, just like architectural mismatch, but it can also precipitate the insidious runtime errors mentioned before.

These assumptions can take two forms. *Provides* assumptions describe the services a component provides to

its users or clients. *Requires* assumptions detail the services or resources that a component must have in order to correctly function. Mismatch between two components occurs when their provides and requires assumptions do not match up.

What can you do about interface mismatch? Besides changing your requirements so that yesterday's bug is today's feature (which is often a viable option), there are three things:

- Avoid it by carefully *specifying* and inspecting the components for yoursystem.
- Detect those cases you have not avoided by careful *qualification* of thecomponents.
- Repair those cases you have detected by *adapting* thecomponents.

The rest of this section will deal with techniques for avoiding, detecting, and repairing mismatch. We begin with repair.

TECHNIQUES FOR AVOIDING INTERFACE MISMATCH

One technique for avoiding interface mismatch is to undertake, from the earliest phases of design, a disciplined approach to specifying as many assumptions about a component's interface as feasible. Is it feasible or even possible to specify all of the assumptions a component makes about its environment, or that the components used are allowed to make about it? Of course not. Is there any evidence that it is practical to specify an important subset of assumptions, and that it pays to do so? Yes. The A-7E software design presented in Chapter 3 partitioned the system into a hierarchical tree of modules, with three modules at the highest level, decomposed into about 120 modules at the leaves. An interface specification was written for each leaf module that included the access programs (what would now be called methods in an object-based design), the parameters they required and returned, the visible effects of calling the program, the system generation parameters that allowed compile-time tailoring of the module, and a set of assumptions (about a dozen for each module).

Component-Based Design as Search

Since component capabilities and liabilities are a principle source of architectural constraint in system development, and since systems use multiple components, component-based system design becomes a search for compatible *ensembles* of off-the-shelf components that come the closest to meeting system objectives. The architect must determine if it is feasible to integrate the components in each ensemble and, in particular, to evaluate whether an

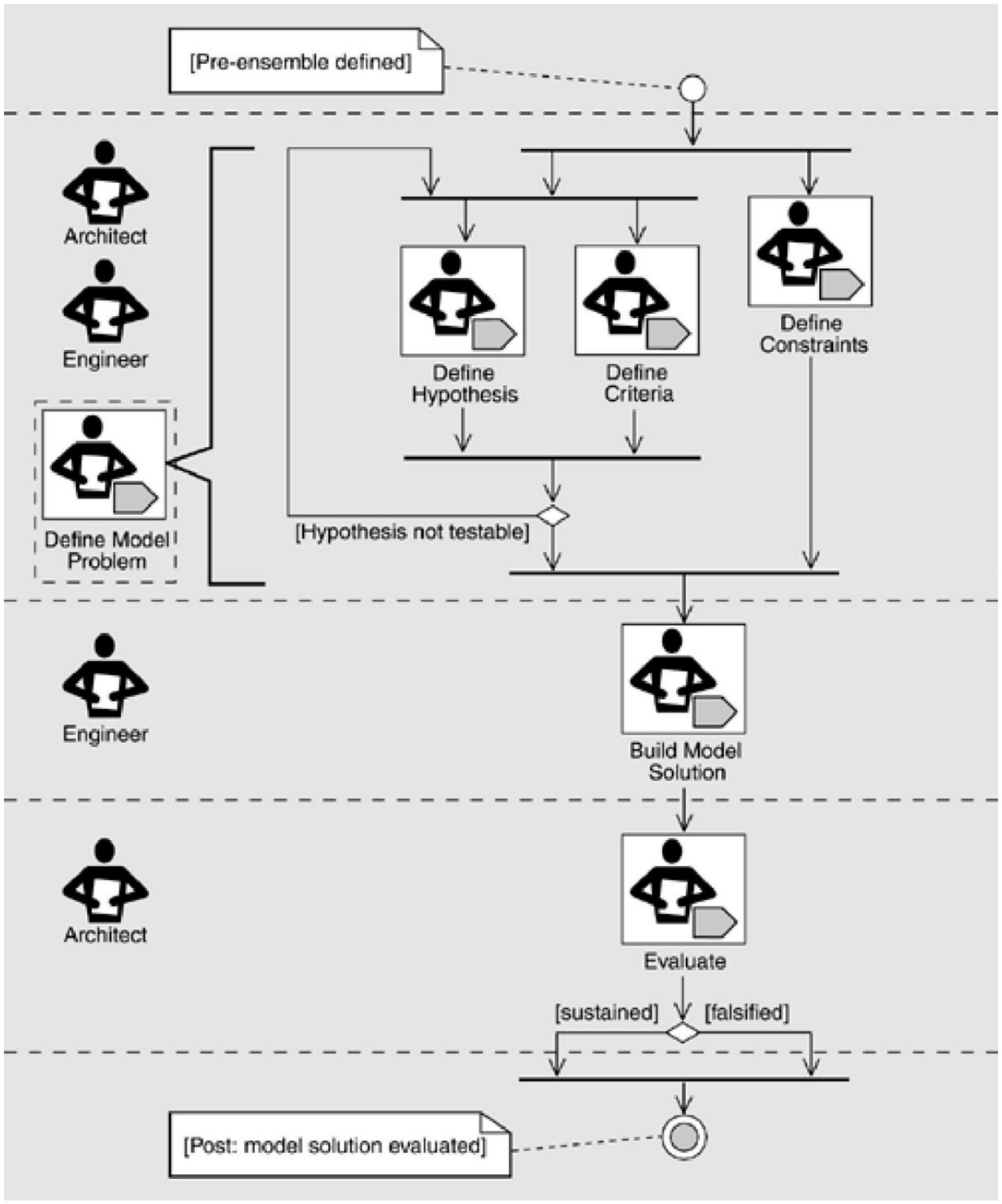
ensemble can live in the architecture and support system requirements.

In effect, each possible ensemble amounts to a continued path of exploration. This exploration should initially focus on the feasibility of the path to make sure there are no significant architectural mismatches that cannot be reasonably adapted. It must also take into account the feasibility of the repair and the residual risk remaining once the repair is completed.

An illustration of the model problem work flow is shown in Figure 18.1. The process consists of the following six steps that can be executed in sequence:

1. The architect and the engineers identify a *design question*. The design questioninitiates the model problem, referring to an unknown that is expressed as ahypothesis.
2. The architect and the engineers define the *starting evaluation criteria*. These criteria describe how the model solution will support or contradict thehypothesis.
3. The architect and the engineers define the *implementation constraints*. The implementation constraints specify the fixed (inflexible) part of the design context that governs the implementation of the model solution. These constraints might include such things as platform requirements, component versions, and businessrules.
4. The engineers produce a *model solution* situated in the design context. The model solution is a minimal application that uses only the features of a component(or components) necessary to support or contradict thehypothesis.
5. The engineers identify *ending evaluation criteria*. Ending evaluation criteria include the starting set plus criteria that are discovered as a by-product of implementing the model solution.
6. The architect performs an *evaluation* of the model solution against the ending criteria. The evaluation may result in the design solution being rejected or adopted.

Figure 18.1. Model problem work flow



ASEILM Example

Our example centers around a Web-based information system developed at the Software Engineering Institute (SEI) for automating administrative interactions between SEI and its transition partners. The Automated SEI Licensee Management (ASEILM) system was created with the following objectives:

- To support the distribution of SEI-licensed materials, such as courses and assessment kits, to authorizedindividuals

- To collect administrative information forassessments
- To graphically present revenue, attendance, and other information about SEI licensed materials
- To track course attendance and royalties due toSEI

ASEILM must support the following multiple user types, each with varying authorization to perform system functions:

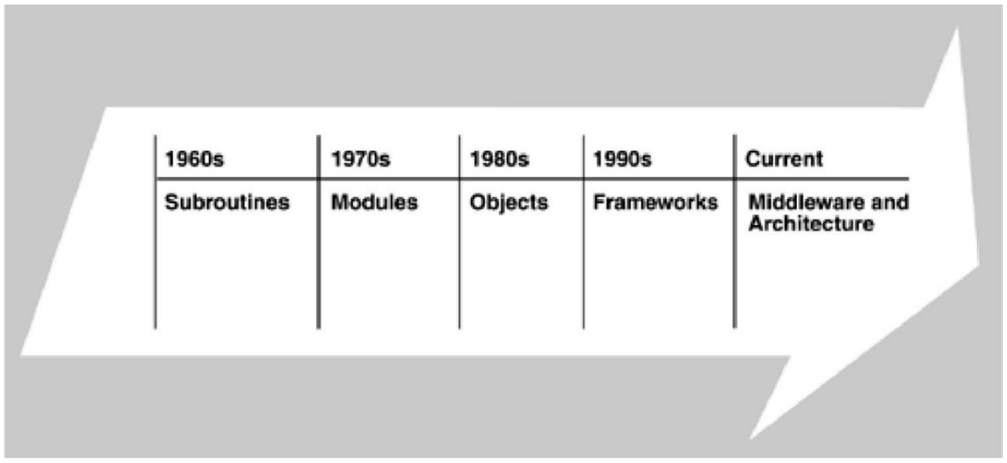
- Course instructors can input course attendee lists, maintain contact information,and download course materials.
- Lead assessors can set up assessments, input assessment information, anddownload assessment kits.
- SEI administrators can maintain lists of authorized instructors and lead assessors, as well as view or edit any information maintained by thesystem.

Quality Attribute	Requirement
Functionality	Provide Web-based access to a geographically dispersed customer base
Performance	Provide adequate performance to users running overseas on low-bandwidth connections (i.e., download times in tens of minutes, not hours)
Compatibility	Support older versions of Web browsers including Netscape 3.0 and Internet Explorer 3.0
Security	Support multiple classes of users and provide an identification and authorization scheme to allow users to identifythemselves
Security	Provide commercial-grade secure transfer of data over theInternet

Software Architecture in theFuture

The history of programming can be viewed as a succession of ever-increasing facilities for expressing complex functionality. In the beginning, assembly language offered the most elementary of abstractions: exactly where in physical memory things resided (relative to the address in some base register) and the machine code necessary to perform primitive arithmetic and move operations. Even in this primitive environment programs exhibited architectures: Elements were blocks of code connected by physical proximity to one another or knitted together by branching statements or perhaps subroutines whose connectors were

of branch-and-return construction. Early programming languages institutionalized these constructs with connectors being the semicolon, the goto statement, and the parameterized function call. The 1960s was the decade of the subroutine.



Growth in the types of abstraction available over time

The Architecture Business Cycle Revisited

In this context, we can now identify and discuss four different versions of the ABC that appear to have particular promise in terms of future research:

- The simplest case, in which a single organization creates a single architecture fora single system
- One in which a business creates not just a single system from an architecture but an entire product line of systems that are related by a common architecture and a common assetbase
- One in which, through a community-wide effort, a standard architecture or reference architecture is created from which large numbers of systemsflow

- One in which the architecture becomes so pervasive that the developing organization effectively becomes the world, as in the case of the World WideWeb

Creating an Architecture

we emphasized the quality requirements for the system being built, the tactics used by the architect, and how these tactics were manifested in the architecture. Yet this process of moving from quality requirements to architectural designs remains an area where much fruitful research can be done. The design process remains an art, and introducing more science into the process will yield large results.

Answers to the following questions will improve the design process:

- *Are the lists of quality attribute scenarios and tacticscomplete?*
- *How are scenarios and tacticscoupled?*
- *How can the results of applying a tactic bepredicted?*
- *How are tactics combined intopatterns?*
- *What kind of tool support can assist in the designprocess?*
- *Can tactics be "woven" intosystems?*

Architecture within the LifeCycle

Although we have argued that architecture is *the* central artifact within the life cycle, the fact remains that a life cycle for a particular system comprises far more than architecture development. We see several areas ripe for research about architecture within the lifecycle:

- *Documentation within a toolenvironment*
- *Software architecture within configuration managementsystems.*
- *Moving from architecture tocode.*

The Impact of Commercial Components

the capabilities and availability of commercial components are growing rapidly. So too are the availability of domain-specific architectures and the frameworks to support them, including the J2EE for information technology architectures. The day is coming when domain-specific architectures and frameworks will be available for many of today's common domains. As a result, architects will be concerned as much with constraints caused by the chosen framework as by green-field design.

Not even the availability of components with extensive functionality will free the architect from the problems of design, however. The first thing the architect must do is determine the properties of the used components. Components reflect architectural assumptions, and it becomes the job of the architect to identify them and assess their impact on the system being designed. This requires either a rich collection of attribute models or extensive laboratory work, Determination of the quality characteristics of components and the associated framework is important for design using externally constructed components. We discussed a number of options with J2EE/EJB in Chapter 16 and the performance impact of each. How will the architect know the effect of options that the framework provides, and, even more difficult, the qualities achieved when the architect has no options? We need a method of enumerating the architectural assumptions of components and understanding the consequences of a particular choice.

Finally, components and their associated frameworks must be produced and the production must be designed to achieve desired qualities. Their designers must consider an industry-wide set of stakeholders rather than those for a single company. Furthermore, the quality attribute requirements that come from the many stakeholders in an industry will likely vary more widely than the requirements that come from the stakeholders of a single company.